RĪGAS TEHNISKĀ UNIVERSITĀTE

Datorzinātnes un informācijas tehnoloģijas fakultāte

Lietišķo datorsistēmu institūts

Aleksandrs SUHORUKOVS

Doktora studiju programmas „Datorsistēmas“ doktorants

DATORSISTĒMU TESTĒŠANAS AUTOMATIZĀCIJAS METODES, RĪKI UN EFEKTIVITĀTE

Promocijas darbs

Zinātniskais vadītājs

Dr. sc. ing., profesore

L. ZAICEVA

Rīga 2011

ANOTĀCIJA

Promocijas darbs veltīts datorsistēmu testēšanas automatizācijas metodēm un rīkiem, kā arī to efektivitātes novērtēšanai.

Darbā tiek izskatīti aktuālie automatizētās testēšanas attīstības virzieni, tiek klasificēti populāri testu automatizācijas rīki. Tiek izstrādāts vienots automatizētās testēšanas modelis, kas augstā līmenī apraksta automatizēto testēšanu, atspoguļojot potenciāli automatizējamas programmatūras testēšanas aktivitātes. Tiek izstrādāts matemātiskais automatizētās testēšanas efektivitātes novērtēšanas modelis, kas ļauj panākt racionālu ierobežota testēšanas laika izmantošanu. Tiek izstrādātas automatizētas testu komplektu ģenerēšanas un adaptīvas veiktspējas testēšanas metodes, uz kuru pamata tiek izstrādāti rīki, kuri ir aprobēti reālos projektos.

Darbs sastāv no ievada, sešām nodaļām un nobeiguma.

Pirmajā daļā tiek sniegts ieskats automatizētās testēšanas vēsturē, tiek apskatīti mūsdienās aktuālie testu ģenerēšanas metožu pētījumu virzieni, kā arī tiek piedāvāta testu automatizācijas rīku klasifikācija, pēc kuras tiek klasificēti 32 populārie rīki.

Otrajā daļa tiek sniegts pārskats par testu automatizācijas rīku un testu programmatūras izstrādes procesiem, kā arī tiek aprakstīts izstrādātais vienots automatizētās testēšanas modelis.

Trešajā daļā tiek aprakstīts izstrādātais automatizētās testēšanas efektivitātes novērtēšanas modelis.

Ceturtajā daļā tiek aprakstīti izstrādātie automatizētās testu komplektu ģenerēšanas risinājumi.

Piektajā daļā tiek aprakstīts izstrādātais veiktspējas testēšanas rīks un izstrādātā adaptīvā veiktspējas testēšanas metode.

Sestajā daļā tiek īsi raksturota ceturtajā un piektajā daļā aprakstīto risinājumu ieviešanas un izmantošanas pieredze reālos projektos.

Darbs satur 130 lpp. teksta, 30 attēlus, 8 tabulas un 130 bibliogrāfiskos avotus.

ABSTRACT

The present Thesis is dedicated to the methods and tools of computer systems testing automation as well as estimation of their efficiency.

Topical directions of automated testing evolution are overviewed, popular test automation tools are classified. Unified automated testing model is developed allowing to describe automatable activities of software testing on a high level. Mathematical model of automated testing efficiency estimation is developed allowing to achieve rational usage of a testing time. Automated test suite generation methods and adaptive performance testing methods are developed, providing the basis for developed tools, which were applied in the real testing projects.

The Thesis consists of the introduction, six chapters and the conclusion.

The first chapter describes historical evolution of automated testing, overviews several directions of test generation methods research and introduces developed classification of test automation tools used to classify 32 popular tools.

The second chapter describes the development process of test automation tools and testware. Developed unified automated testing model is also described.

The third chapter introduces the developed automated testing efficiency evaluation model.

The fourth chapter describes developed solution for test suite generation.

The fifth chapter describes developed performance testing tool and adaptive performance testing method.

The sixth chapter overviews introduction and usage experience of the developed solutions described in the chapter four and five in real projects.

The Thesis consists of 130 pages of text, 30 figures, 8 tables and 130 bibliographical references.

SATURS

IEVADS

Pētījuma motivācija

Pēdējos laikos arvien aktuālāks kļūst jautājums par programmatūras testu automatizāciju. Attīstoties tehnoloģijām, sistēmas kļūst lielākas un sarežģītākas, bet izstrādes cikli — ātrāki. Līdz ar to pieaug arī testu skaits un nepieciešamība pēc ātras testu izpildes. Šādos apstākļos, un ņemot vērā, ka programmatūras testēšana ir laikietilpīgs process, testēšanas efektivitāti var palielināt automatizētie testi, kuru izpilde prasa mazāk cilvēka laika, salīdzinot ar manuālo testu izpildi.

Automatizācijas lietderību nosaka nepieciešamība pēc testu daudzkārtējas atkārtošanas, piemēram, pārbaudot vairākus ieejas datus, pārbaudot sistēmu pēc veiktām izmaiņām utt. Testēšanas efektivitāti var ievērojami palielināt, ja šāda veida atkārtojošās darbības uzticēt veikt rīkam.

Automatizētais tests nekad nenogurst, nekļūst mazāk uzmanīgs un vienmēr precīzi veic paredzētu uzdevumu, turpretim manuālajai testēšanai piemīt vairāki citādi grūti novēršami trūkumi:

- tā ir lēnāka tādā nozīmē, ka parādoties jaunam sistēmas būvējumam, ir jāpatērē daudz laika, lai paveiktu visus galvenos testus, kas dotu priekšstatu par konkrētā būvējuma kvalitāti, bet automatizētie testi, atšķirībā no manuālajiem, izpilda testus ievērojami ātrāk, kā arī ir darbināmi ārpus parastā darba laikā, piemēram, naktīs un brīvdienās;
- galvenie testi bieži nemainās starp sistēmas būvējumiem un tie vienmēr jāatkārto pēc programmatūrā veiktām izmaiņām, kas ir rutīnas darbs un nav labākais kvalificēta personāla laika izmantošanas veids;
- trūkums, kas izriet no iepriekšēja — viena un tā paša testa atkārtošana, atstāj mazāk laika jaunu testu izstrādei un veikšanai, kas palēnina testu pārklājuma palielināšanu;
- vienu un to pašu testa darbību daudzkārtēja atkārtošana nogurdina testētāju un samazina uzmanības līmeni, kā rezultātā palielinās iespējas nepamanīt defektus.

Neskatoties uz automatizēto testu priekšrocībām, tiem ir arī vērā ņemami trūkumi, salīdzinot ar manuālajiem:

- automatizētais tests dara tikai to, kas tam ir jādara, un pārbauda tikai to, kas jāpārbauda, tas nevar pamanīt acīmredzamu defektu, ja attiecīga pārbaude nav paredzēta;

- automatizētais tests spēj pārbaudīt mazāk testējamās sistēmas īpašību, nekā cilvēks, piemēram, tas nevar novērtēt lietošanas ērtumu, skaistumu un citus neformalizējamus aspektus;
- kaut arī automatizētā testa izpilde aizņem salīdzinoši maz laika, tā sagatavošanai nepieciešamais laiks var ievērojami pārsniegt manuālā testa sagatavošanas laiku.

Tā kā gan manuālajai, gan automatizētajai testēšanai ir gan priekšrocības, gan trūkumi, jautājums par to, kura no metodēm ir jālieto konkrētajā gadījumā, nav triviāls. Divas galējības — testu automatizācijas iespēju ignorēšana un centieni automatizēt visus testus — nav racionālas gan no darbietilpības, gan no testēšanas kvalitātes viedokļa [82]. Piemērota līdzsvara noteikšana ir problēma, kurai pagaidām nav viennozīmīga risinājuma.

Testēšanas automatizācijas tehnoloģijas mūsdienās ir labi attīstītas vairākos virzienos un piemērojamas dažādu, pat specifisku, sistēmu testēšanai [118]. Ir pieejami daudzi rīki funkcionālās un veiktspējas testēšanas automatizācijas nodrošināšanai. Vienībtestēšanas risinājumi ir pieejami gandrīz vai katrai pastāvošai programmēšanas valodai [49]. Rīku dažādība nodrošina plašāku automatizācijai piemērotu testu klāstu, bet tajā pašā laikā apgrūtina piemērotākā rīka izvēles uzdevumu.

Daudzas organizācijas gan Latvijā, gan pasaulē ir mēģinājušas, mēģina vai jau lieto testēšanas automatizācijas iespējas. Taču joprojām pastāv daudzi atšķirīgi viedokļi par to, kā jāveic testu automatizācija, un kādai ir jābūt automatizācijas vietai testēšanā un vispār programmatūras izstrādē [88, 107]. Viedokļu dažādības dēļ ir grūti novērtēt kādas metodes un kādā apjomā jālieto konkrēta projekta vajadzībām. Tas var novest pie neefektīvas testēšanas plānošanas, kad vai nu no automatizācijas atsakās vispār, vai arī automatizē tādus testus, kuru manuālā izpilde būtu izdevīgāka.

Kad testēšanas automatizācija jau tiek pielietota, vai arī kad to tikai plāno ieviest, ir būtiski novērtēt vai prognozēt tās izmantošanas efektivitāti. Pagaidām nepastāv automatizētās testēšanas efektivitātes novērtēšanas metode, kas ļautu salīdzināt izvēlētas automatizācijas stratēģijas efektivitāti ar līdzvērtīgās manuālās testēšanas stratēģiju, vai ar citām alternatīvām automatizācijas stratēģijām.

Augstāk minētās problēmas nosaka šīs tēmas aktualitāti un nepieciešamību pēc padziļinātiem pētījumiem datorsistēmu testēšanas automatizācijas metožu un efektivitātes jomā.

Darba mērķis un uzdevumi

Promocijas darba mērķis ir, balstoties uz testēšanas metožu analīzi un testu automatizācijas rīku pētīšanu, izstrādāt testēšanas automatizācijas metodes, kas ļautu panākt testēšanai veltītā laika ekonomiju un/vai labāku testēšanas kvalitāti, un automatizēto testu efektivitātes novērtēšanas modeli, kā arī aprobēt izstrādātās metodes un rīkus reālos projektos.

Lai sasniegtu izvirzīto mērķi, ir jārisina šādi uzdevumi:

1) izpētīt dažādu testēšanas aktivitāšu automatizācijas esošas metodes un klasificēt mūsdienās pieejamus testu automatizācijas rīkus;
2) izstrādāt automatizētās testēšanas modeli, kas ļautu identificēt potenciāli automatizējamos procesus programmatūras testēšanā;
3) izstrādāt automatizēto testu efektivitātes novērtēšanas modeli;
4) izstrādāt testēšanas automatizācijas risinājumus ar augstākiem efektivitātes rādītājiem salīdzinājumā ar esošiem risinājumiem;
5) aprobēt izstrādātos risinājumus reālos projektos.

Formulētie uzdevumi atspoguļo darba struktūru. Katram uzdevumam tiek veltīta atsevišķa nodaļa, bet izstrādātajiem risinājumiem — divas nodaļas. Kopējie secinājumi ir ievietoti nobeigumā.

Pētījuma metodika, zinātniskā novitāte, darba praktiskā nozīme

Pētījuma joma ir datorsistēmu testēšana, izmantojot programmatūras rīkus, kuri automatizē testēšanas aktivitāšu izpildi.

Pētījuma priekšmets ir automatizētās datorsistēmu testēšanas metodes, efektivitāte un rīki, kas to nodrošina.

Pētījuma metodika ir kopu teorija, grafu teorija, algoritmu projektēšana un analīze.

Darba rezultāta jaunieguvumi ir šādi:

- izstrādāts automatizēto testu efektivitātes novērtēšanas modelis;
- izstrādātas automātiskās testu komplektu ģenerēšanas metodes, balstoties uz vienkāršo un salikto stāvokļu modeļiem;
- izstrādāts veiktspējas testēšanas adaptīvā slodzes plānotāja modelis un piedāvāts tā realizācijas algoritms.

Darba rezultātu galvenā praktiskā vērtība ir saistīta ar izstrādātajām testu komplektu ģenerēšanas metodēm un veiktspējas testēšanas adaptīvā slodzes plānotāja modeli, kas pie noteiktiem nosacījumiem izrādās efektīvāki par citām metodēm. Šīs metodes tika realizētas autora izstrādātajos rīkos, kuri tika sekmīgi pielietoti četrpadsmit reālos testēšanas projektos dažādos uzņēmumos.

Par darba rezultātiem ir ziņots šādās starptautiskajās konferencēs:

1. 50th International Scientific Conference of Riga Technical University, 12-16 October 2009, Riga, Latvia.
2. 13th East-European Conference on Advances in Databases and Information Systems (ADBIS 2009), Workshop on Intelligent Educational Systems and Technology-enhanced Learning (INTEL-EDU), 7 September 2009, Riga, Latvia.
3. International Conference on Advanced Learning Technologies (ICALT-2009), 15-17 July 2009, Riga, Latvia.
4. 49th International Scientific Conference of Riga Technical University, 13-15 October 2008, Riga, Latvia.
5. International Conference on Information Technologies (InfoTech-2008), 19-20 September 2008, Varna, Bulgaria.
6. International Conference on Engineering Education & Research (ICEER 2007), 2-7 December 2007, Melbourne, Australia.
7. 48th International Scientific Conference of Riga Technical University, 11-13 October 2007, Riga, Latvia.

Darba rezultāti ir publicēti šādos starptautiskajos izdevumos:

1. Sukhorukov A. Automated Test Chaining Based on Compound States // Proceedings of the International Conference on Information Technologies (InfoTech-2010). – Sofia, Bulgaria: Publishing House of Technical University, 2010. – pp. 127–132.
2. Sukhorukov A. Self-Directed Performance Testing // Scientific Journal of Riga Technical University. Series 5 – Computer Science. Applied Computer Systems. – 2010. – Vol. 43. – pp. 84–89.
3. Сухоруков А. Целенаправленное обучение на примере модели и классификатора инструментов автоматизации тестирования ПО // Образовательные технологии и общество. – Казань, Татарстан, РФ: Казанский государственный технологический университет, 2010. – Том 13, № 1. – стр. 370–377.
4. Sukhorukov A. Test Case Generation for Validation of E-Learning Course // Advances in Databases and Information Systems, 13th East-European Conference, ADBIS 2009

Associated Workshops and Doctoral Consortium. Local Proceedings. – Riga, Latvia: Riga Technical University, 2009. – pp. 230–237.

5. Sukhorukov A. Architecture for Automated Validation of E-Learning Courses // Proceedings of the Ninth IEEE International Conference on Advanced Learning Technologies (ICALT-2009). – Washington, DC, USA: IEEE Computer Society, 2009. – pp. 152–153.
6. Sukhorukov A. Problems of Test-Driven Aspect-Oriented Development // Scientific Proceedings of Riga Technical University. Series 5 – Computer Science. Applied Computer Systems. – 2009. – Vol. 38. – pp. 180–186.
7. Sukhorukov A. Performance testing tool Picus // Proceedings of the 22nd International Conference on Systems for Automation of Engineering and Research (SAER-2008). – Bulgaria: King, 2008. – pp. 165–172.
8. Sukhorukov A., Zaitseva L. Automated test state management framework // Scientific Proceedings of Riga Technical University. Series 5 – Computer Science. Applied Computer Systems. – 2008. – Vol. 34. – pp. 215–224.
9. Sukhorukov A., Zaitseva L. Applicability of Automated Testing to E-Learning Software Validation // Proceedings of the International Conference on Engineering Education & Research (ICEER 2007). – Melbourne, Australia: Victoria University, 2007. – pp. 1–6.

Izmantoto terminu definīcijas

Darbā, kur vien iespējams, tiek izmantota standarta terminoloģija no atzītajiem avotiem tādiem kā Angļu-latviešu-krievu informātikas vārdnīca [7], IEEE Standard Glossary of Software Engineering Terminology [62], ISTQB Glossary of Testing Terms [124] un citiem.

Automatizētais tests (*Automated test*) Tests, kura izpilde var notikt automātiski bez cilvēka līdzdalības vai ar minimālu līdzdalību.

Automatizētā testēšana (*Automated testing*) Testēšana, kurā viena vai vairākas aktivitātes ir automatizētas, t.i., ar noteiktu rīku atbalstu, var tikt veiktas bez cilvēka līdzdalības vai ar minimālu līdzdalību.

Automatizētās testēšanas rīks (*Automated testing tool*) Rīks, kas tiek izmantots automatizētajā testēšanā, lai cilvēka vietā veiktu rutīnas darbības.

Defektpunkts (*Defect point*) Defekta vai defektu kopas smaguma mērvienība.

Defekts (*Defect*) Sistēmas nepilnība, neatbilstība prasībām vai testa sagaidāmajam rezultātam.

Manuālais tests (*Manual test*) Tests, kura izpilde nav automatizēta un tas ir jāizpilda cilvēkam — testētājam.

Manuālā testēšana (*Manual testing*) Testēšana, kuras aktivitātes nav vai ir vāji automatizētas.

Testpiemērs (*Test case*) Ievaddatu, izpildes noteikumu un sagaidāmo rezultātu kopa, izstrādāta lai pārbaudītu testējamā objekta atbilstību iepriekš definētajām prasībām.

Testa dati (*Test data*) Dati, ko izmanto testa skripts, lai izpildītu testpiemēru. Iekļauj ievaddatus, sagaidāmos rezultātus, kā arī informāciju par testa pirmsnosacījumiem un pēcnosacījumiem.

Testa skripts (*Test script*) Programmatūra, kas nodrošina testa automātisku izpildi. Plašākā nozīmē var būt arī manuālo darbību apraksts, kas ir jāveic, pildot testu.

Testēšana (*Testing*) Process, kas aptver visas produkta dzīvescikla aktivitātes, ar mērķi novērtēt tā atbilstību prasībām un piemērotību nolūkam, kā arī atklāt defektus.

Testēšanas automatizācija (*Testing automation*) Vienas vai vairāku testēšanas procesa aktivitāšu pilnveidošana, daļēji vai pilnībā pārliekot to veikšanu programmatūras rīkiem.

Testēšanas efektivitāte (*Testing efficiency*) Pakāpe, kādā tiek sasniegti testēšanas mērķi, attiecībā pret tās izmaksām.

Testēšanas process (*Testing process*) Tiek izmantots kā termina „Testēšana“ sinonīms.

Testēšanas rezultativitāte (*Test effectiveness*) Pakāpe, kādā tiek sasniegti testēšanas mērķi.

Testēšanas rīks (*Testing tool*) Programmatūra, kas ir izmantojama testēšanas procesā, lai atvieglotu vai automatizētu dažādu aktivitāšu veikšanu.

Tests (*Test*) Viena vai vairāku testpiemēru kopa.

Testu automatizācija (*Test automation*) Automatizēto testu izveides process.

Testu automatizācijas rīks (*Test automation tool*) Rīks, kas atbalsta automatizēto testu izveidi un izpildi.

Testu komplekts (*Test suite*) Sakārtota testu kopa, kurā katra testa pēcnosacījumi tiek izmantoti kā nākamā testa pirmsnosacījumi.

Testu kopa (*Test set*) Vairāki testi, kurus ir lietderīgi izskatīt kā vienu veselumu.

Testu programmatūra (*Testware*) Visa programmatūra, kas tiek izstrādāta testēšanas procesa ietvaros, piemēram, testu skripti, testpiemēri, testu komplekti utt.

Testu ģenerēšana (*Test generation*) Automatizēta testa datu vai skriptu izveide atbilstoši uzdotajiem nosacījumiem, parasti vairākiem testpiemēriem.

1. AUTOMATIZĀCIJA PROGRAMMATŪRAS TESTĒŠANĀ

1.1. Automatizācija testēšanas attīstībā

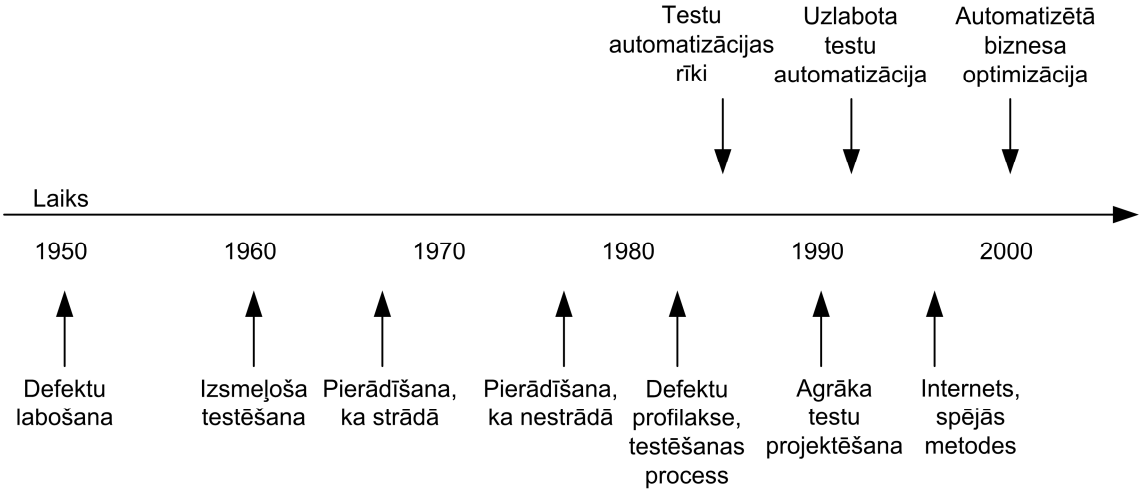
Jau pirms vairāk kā divdesmit gadiem tika veikti mēģinājumi sadalīt programmatūras testēšanas attīstību vēsturiskajos posmos. Piemēram, 1988. gadā D. Gelperin un B. Hetzel [44] piedāvāja sadalījumu, kas ir atspoguļots 1.1. tabulā.

1.1. tabula

Programmatūras testēšanas attīstības posmi (pēc D. Gelperin un B. Hetzel)

Gadi	**Posms**
… – 1956	Orientēta uz atkļūdošanu (*debugging*)
1957 – 1978	Orientēta uz demonstrēšanu (*demonstration*)
1979 – 1982	Orientēta uz izpostīšanu (*destruction*)
1983 – 1987	Orientēta uz novērtēšanu (*evaluation*)
1988 – …	Orientēta uz profilaksi (*prevention*)

W. E. Lewis [80] piedāvā jau mūsdienīgāku skatu uz testēšanas vēsturi. Viņa piedāvātā shēma ir atspoguļota 1.1. attēlā.



1.1. att. Programmatūras testēšanas vēsture (pēc W. E. Lewis)

Sākotnēji testēšana netika uztverta kā atsevišķa aktivitāte, testēšana un atkļūdošana nebija atdalāmi. Taču ar laiku tika apzināts, ka programmatūras izstrādei ir daudz līdzīga ar citu tehnisko sistēmu izstrādi, un līdz ar to klasiskās inženierijas metodes var tikt izmantotas arī izstrādājot programmatūru. Nozīmīgs solis programminženierijas virzienā tika sperts 1970. gadā, kad W. Royce publicēja rakstu par lielo datorsistēmu izstrādes modeli [106], kas vēlāk kļuva pazīstams kā ūdenskrituma modelis [102].

Ūdenskrituma modelī fāzes-aktivitātes, tādas kā analīze, projektēšana, kodēšana, utt., seko viena aiz otras. Katras fāzes beigas novērtē pēc noteiktiem kritērijiem, un beidzoties vienai fāzei, sākas nākamā. Kaut arī starp fāzēm pastāv stingras robežas, ir iespējams atgriezties uz iepriekšējo fāzi, ja tiek atklātas kļūdas, kas tika pieļautas iepriekšējā fāzē.

Testēšana ūdenskrituma modelī ir fāze, kas seko aiz kodēšanas. Tātad testēšana tiek uzsākta tikai tad, kad programma jau ir gatava. Testēšanai šajā modelī ir ierobežota loma — pārbaudīt, vai produkts jau ir gatavs ekspluatācijai. Gadījumā ja testēšanas gaitā tiek atklātas problēmas, notiek atgriešanās uz iepriekšējo fāzi, un seko problēmu labošana. Pēc labošanas process atkal atgriežas testēšanas fāzē, kad labotās problēmas tiek pārtestētas.

Testēšanas vieta un loma ūdenskrituma modelī sasaucās ar testēšanas attīstības stāvokli 1970. gados (uz demonstrēšanu orientētā testēšana), kad galvenais testēšanas uzdevums bija apliecināt, ka programma darbojas. Taču praksē notika tā (un notiek arī mūsdienās, ja testēšanu atliek līdz projekta beigām), ka sākot testēšanu, ātri atrod daudz defektu. Tas parasti notiek vairākās iterācijās — pēc defektu izlabošanas tiek atrasti jauni defekti, kuri tiek ieviesti, labojot vecus. Tāpēc ūdenskrituma modelis tuvojoties beigu fāzēm var pārvērsties par iteratīvo testēšanas-labošanas procesu.

Programmatūras defektu neizbēgamības apzināšana un to novēršanas nepieciešamība mainīja attieksmi pret testēšanu. Līdz ar to 1970. gadu beigās tās loma ievērojami izauga un par būtiskāko testēšanas uzdevumu sāka uzskatīt programmatūras defektu atklāšanu. Par veiksmīgajiem testiem sāka uzskatīt tādus, kas atklāj defektus. Kļuva skaidrs, ka defektu meklēšana nav triviāls uzdevums, un prasa specifiskas zināšanas un prasmes.

Pie nozīmīgiem šī posma notikumiem var pieskaitīt šādus:

- 1976. gadā M. E. Fagan piedāvā programmatūras formālo inspekciju metodi [36], kas vēlāk pēc vairākiem uzlabojumiem tika iekļauta tekošajā starptautiskajā standartā IEEE 1028-2008 [61] un mūsdienās tiek uzskatīta par vienu no statiskās testēšanas metodēm [47];

- 1979. gadā G. J. Myers publicē grāmatu „The Art of Software Testing“ [91], kur pirmo reizi testēšana tiek atdalīta no atkļūdošanas, tādējādi testēšana tiek pasludināta par atsevišķu disciplīnu;
- strauji attīstās automatizētās testēšanas metodes [66, 73], šajā virzienā strādā arī Latvijas zinātnieki [18].

Domas par to, ka testēšanai būtu pēc iespējas agrāk jāiesaistās programmatūras izstrādes dzīvesciklā, skanēja jau 1970. gadu beigās, bet tikai 1980. gados tā ideja guva plašu atzinumu. Tad arī tika veikti sistemātiski pētījumi par šo tēmu [3, 20]. Programmatūras defektu izcelsmes pētījumi, piemēram [13], liecināja, ka lielākoties defekti programmatūrā tiek pielaisti prasību analīzes un projektēšanas fāzēs. No tā izrietēja, ka nav efektīvi atlikt testēšanu līdz brīdim, kad tiks beigta kodēšana, bet ir jānovērtē arī citu izstrādes fāžu produkti. Gūst popularitāti programmatūras kvalitātes nodrošināšana kā atsevišķa disciplīna [24]. Korektums vairs nav vienīgā īpašība, kas raksturo kvalitāti un kas testēšanā jānovērtē, ir būtiskas arī citas: efektivitāte, modificējamība, testējamība utt.

Agra testēšanas iesaistīšana un nepieciešamība pēc ātras novērtēšanas noveda pie iteratīvajiem izstrādes modeļiem. Tie ļāva panākt ātrāku risku identificēšanu un mazināšanu. Katra iterācija satur vairākas vai visas no ūdenskrituma modelī esošajām aktivitātēm. 1988. gadā B. W. Boehm piedāvāja programmatūras izstrādes spirāles modeli [19]. Galveno uzsvaru liekot uz agrāku risku mazināšanu, projekta sākumā iterācijas ir mazākas (darbietilpības un izmaksu ziņā) nekā nākamās. Iespējamas problēmas ir jāpamana un jāmēģina novērst pašā sākumā, kad tas ir lēti.

Iteratīvās izstrādes gadījumā kļuva aktuāla testu atkārtošana — regresīvā testēšana. Jo vairāk ir iterāciju, jo ticamāk, ka būs vairakkārt jāatkārto tie paši testi, lai pārliecinātos, ka esošas izstrādājamā produkta īpašības nav sabojātas. 1980. gadu vidū sāk parādīties automatizētās testēšanas rīki, balstīti uz ierakstīšanas/atspēlēšanas mehānismu, piemēram Assay [79] un Encore [75]. Tie bija spējīgi ierakstīt cilvēka darbības, ko viņš veic ar programmu, un pašas programmas darbības rezultātus, bet vēlāk — atkārtot tās pašas darbības un salīdzināt iegūtos rezultātus ar iepriekš ierakstītajiem. Tas bija ievērojams solis uz priekšu, jo ļāva automatizēt programmas testēšanu, izmantojot melnās kastes pieeju. Pirms tam automatizētās testēšanas rīki bija balstīti uz programmas modeļu vai pirmkoda analīzi un simbolisko izpildi, un tāpēc bija maz piemēroti testēšanai augstākajos līmeņos — sistēmtestēšanai un akcepttestēšanai.

Pirmajiem ierakstīšanas/atspēlēšanas rīkiem bija raksturīga problēma, ka automatizēt testu, t.i., ierakstīt testa skriptu, bija iespējams tikai tad, kad pati programma jau bija gandrīz

gatava, vai, vismaz, bija gatava tās lietotāja saskarne. 1990. gadu sākumā, parādījās attīstītāki rīki, kas ierakstīšanas/atspēlēšanas mehānismu kombinēja ar skriptēšanas iespējām, kas ļāva pielāgot automatizēto testu pēc ierakstīšanas, nodrošināt tam darbības loģiku, tādējādi padarot to par sava veida programmu [43]. Tātad šie rīki jau veidoja automatizēto testu skripta formā. Rīki ātri attīstījās, nodrošinot papildus iespējas [80]:

- ar datiem vadāmā testēšana — iespēja atdalīt testa skriptu no testa datiem, tādējādi ar datu tabulu palīdzību nodrošinot iespēju vienam skriptam izpildīt vairākus testpiemērus;
- modularitāte — iespēja veidot mazus, neatkarīgus automatizācijas skriptus, kas atbilst loģiskajām darbībām ar programmu, un kurus kombinējot, tiek nodrošināta testu automatizācija augstākajā līmenī;
- ar atslēgas vārdiem vadāmā testēšana — iespēja piešķirt maziem, neatkarīgiem automatizācijas skriptiem atslēgas vārdus, kurus var specificēt datu tabulās, kā testa datus;
- un šo iespēju kombinācijas.

Pateicoties šīm jaunām īpašībām, rīki nodrošināja arī iespēju projektēt automatizēto testu komplektus pirms bija pieejama testējamā programma, veidojot datu tabulas, atslēgas vārdu tabulas, un skriptējot augstajā līmenī. Tas lielā mērā saskanēja ar 1990. gados valdošo uzskatu par nepieciešamību projektēt testus pēc iespējas agrākajās projekta stadijās.

Tajā pašā laikā strauji gūst popularitāti objektorientētās modelēšanas valodas, it īpaši UML [41], un objektorientētās izstrādes metodoloģijas, tādas kā Unified Process [76], kas balstījās uz modelēšanas valodu izmantošanu, propagandēja uz modeļiem balstīto projektēšanu, izstrādi un arī testēšanu, lielā mērā pamatojoties arī uz testu automatizācijas rīkiem.

1990. gadu beigās kļuva populāras spējās (*agile*) metodes, tādas kā Scrum [105] un Extreme Programming [15]. Kaut arī spējās metodes balstījās uz jau sen zināmām idejām [1], to popularitātes pieaugumu 1990. gadu beigās – 2000. gadu sākumā lielā mērā ietekmēja izstrādātāju neapmierinātība ar „klasisko“ metožu neelastīgumu. Galvenā īpašība, kas atšķir spējās metodes no citām, ir attieksme pret prasību izmaiņām. Spējo metožu pamatā ir princips, ka no izmaiņām nav jāizvairās, jo prasības jebkurā gadījumā mainīsies, un tam ir jābūt gataviem. Pozitīva attieksme pret izmaiņām, rada nepieciešamību pēc īsām izstrādes iterācijām, lai nodrošinātu pasūtītājam ātrāku atgriezenisko saiti. Īsas iterācijas savukārt pieprasa ātras pārtestēšanas iespēju, jo biežo izmaiņu gadījumā ir būtiski ātri pārliecināties, ka nekas, kas agrāk strādāja labi, netiek sabojāts.

Pateicoties spējām metodēm automatizētā testēšana piedzīvoja jaunu uzplaukumu, jo tagad tā kļuva par nepieciešamu, nevis vēlamu izstrādes projekta sastāvdaļu. Manuālā testēšana vairs nevarēja nodrošināt pietiekami labu pārklājumu, jo iterāciju ilgums bija pavisam mazs. No otras puses esošie testu automatizācijas rīki, kas strādāja lietotāja saskarnes līmenī, nespēja nodrošināt automatizēto testēšanu zemākajos līmeņos (vienībtestēšanā un integrācijtestēšanā). Tāpēc sāka strauji attīstīties vienībtestēšanas ietvari, sākumā JUnit [56] Java valodai, bet tad arī līdzīgi ietvari priekš citām valodām, kas tagad ir zināmi kā xUnit ietvaru saime [49]. Vienībtestēšana pati par sevi arī pirms tam tika veikta un bieži arī automatizēti. Jaunie ietvari vienkārši standartizēja vienībtestu projektēšanas, implementēšanas un izpildes metodi.

No testēšanas viedokļa ir lietderīgi atsevišķi apskatīt tādu spējo metodi kā ar testiem vadāmā izstrāde (*test-driven development*) [14].

Ar testiem vadāmās izstrādes īpašība ir tāda, ka automatizētie testi tiek izstrādāti pirms testējamās programmas koda. Tas nodrošina to, ka programmas funkcionalitāte jebkurā brīdī ir pilnībā pārklāta ar testiem, bet izmaiņu ieviešana ir viegla — ja kaut kas tiek sabojāts, testi uzreiz norādīs uz šo faktu.

Šajā izstrādes metodē ir tipiski, ka pilnais darba cikls (testa izveide, kodēšana un nodrošināšana, ka visi testi izpildās sekmīgi) tiek izpildīts dažu minūšu laikā un viena cikla ietvaros vismaz divas reizes tiek izpildīti visi esošie testi. Bez automatizētās vienībtestēšanas šāds temps nebūtu iespējams.

Ar testiem vadāmajai izstrādei ir arī vājas puses — tā var būt grūti savietojama ar dažām izstrādes paradigmām, piemēram ar aspektorientēto programmēšanu [115].

Protams, ne visas spējās metodes balstās uz ar testiem vadāmo izstrādi. Bet tie tik un tā balstās uz automatizētajiem testiem, jo pat ja visu testu palaišana notiek reizi dienā vai nedēļā, manuālā testu izpilde nespētu izturēt šādu tempu.

Aprakstītajos notikumos var ieraudzīt trīs būtiskus posmus, kuros tika iesākta mūsdienās aktuālo automatizētās testēšanas rīku attīstībā:

- 1970. gados — testu ģenerēšanas, simboliskās izpildes rīki;
- 1980. gados — melnās kastes testu automatizācijas, ierakstīšanas/atspēlēšanas rīki;
- 1990. gadu beigās — mūsdienu vienībtestēšanas ietvari.

Mūsdienās testēšanas attīstība turpinās. Arī Latvijā testēšanas jomā tika un tiek veikti pētījumi [9, 45, 128], kopš 2000. gada katru gadu notiek konference „Testēšanas teorija un prakse“, kur notiek diskusijas par Latvijā aktuālajām testēšanas tēmām, un bieži tiek skarti arī automatizētās testēšanas jautājumi.

1.2. Testu ģenerēšanas automatizācijas metodes

Testpiemēru ģenerēšanas metožu izpēte ir sākusies 20. gadsimta 70. gados [18, 73]. Kopš tā laika tika izstrādātas vairākas metodes, sākot ar vienkāršākām, balstītām uz simbolisko programmas izpildi, līdz sarežģītām hibrīdām metodēm, kas balstās uzreiz uz vairākiem modeļu un koda analīzes algoritmiem un ietver mākslīgā intelekta elementus.

Taču joprojām esošu risinājumu un rīku pielietojamība programmatūras izstrādes projektos ir ierobežota. Šo faktu var izskaidrot dažādi [46]. Pirmkārt, programmu analīzes metodes, tādas kā simboliskās izpildes dziņi vai ierobežojumu analizētāji, prasa lielus skaitļošanas resursus. Tādi resursi reāli kļuva pieejami plašam lietotāju lokam tikai pēdējos gados. Otrkārt, skaitļošanas resursu pieaugums padarīja par iespējamu arī citu kvalitātes nodrošināšanas aktivitāšu automatizēšanu, piemēram, modeļu verifikāciju, teorēmu pierādīšanu vai koda caurskati, balstoties uz programmas statisko analīzi. Šādi rīki zināmā mērā varētu kalpot par alternatīvu testu ģenerēšanai.

Testpiemēru ģenerēšanas uzdevumu var raksturot šādi. Sistēmu var uzskatīt par funkciju, kas ievaddatus transformē izvaddatos. Par ievaddatiem varētu uzskatīt failus, ievade no tastatūras, darbības ar peli utt. Izvade varētu būt ģenerētie faili, attēlotās vērtības vai grafika utt. Testpiemēru ģenerēšanas uzdevums ir dotajai programmai atrast tādus ievaddatus vai to komplektus, kas atbilst noteiktiem kritērijiem.

Ideāls scenārijs būtu ģenerēt visu iespējamu ievaddatu komplektu, kas nodrošinātu izsmeļošu programmas testēšanu. Taču šāds uzdevums ir teorētiski paveicams tikai pašām vienkāršākām programmām. Tātad reāli uzģenerēts testpiemēru komplekts nekad nebūs absolūti pilnīgs (pilnās testēšanas nozīmē), bet var būt relatīvi pilnīgs (t.i., pilnīgs attiecībā uz izvirzīto kritēriju). Pilnīguma kritēriji var būt dažāda veida, piemēram:

- daudzums — jāuzģenerē noteikts atšķirīgu testpiemēru daudzums;
- pārklājums — jāuzģenerē testpiemēri, lai tie pārklātu visus stāvokļus, zarus, ceļus, utt.;
- mērķa defekti — jāuzģenerē testpiemēri, kas garantēti atklās noteikta veida defektu, ja tāds ir;
- un vairāki citi.

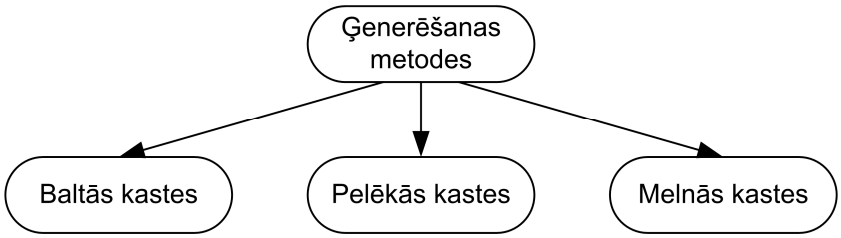
Faktiski, testpiemēru ģenerēšana ir nekas cits, kā testu projektēšanas automatizācija. Šajā nozīmē ir jāatšķir testu projektēšana no testu implementēšanas. Ja projektēšana ir testpiemēru identificēšana ar konkrētiem ievaddatiem, tad implementēšana ir testu skriptu jeb procedūru izveide [47].

Testpiemēru ģenerēšanas metožu ir daudz, līdz ar to būtu lietderīgi tās klasificēt. Metodes var klasificēt dažādi. Šeit aplūkosim četras klasifikācijas atkarībā no:

1) pirmkoda nozīmības;
2) programmas darbināšanas veida;
3) mērķa kritērijiem;
4) modeļu veidiem.

Testpiemēru ģenerēšanas metožu klasifikācija pēc pirmkoda nozīmības

Klasifikācija atkarībā no pirmkoda nozīmības balstās uz klasiski programmatūras testēšanā lietojamām baltās un melnās kastes metodēm (1.2. att.). Tās atšķiras ar to, uz kā pamata testi tiek izvēlēti.



1.2. att. Klasifikācija pēc pirmkoda nozīmības

Baltās kastes metodēs testi tiek izvēlēti, balstoties uz informāciju par programmas implementāciju: ir zināma programmas iekšējā uzbūve un ir pieejams tās pirmkods. Prasību specifikācija netiek lietota.

Testpiemēru ģenerēšanas nolūkiem programmu modelē ar grafiem, piemēram, izmantojot vadības plūsmu (*control flow*) vai datu plūsmas (*data flow*) grafus. Tad atkarībā no izvirzītajiem kritērijiem tiek izvēlēts ceļš no programmas sākuma stāvokļa līdz beigu stāvoklim, un uz tā pamata tiek rēķināti ievaddati, kuru rezultātā programma izies šo ceļu [34].

Melnās kastes metodēs testus izvēlas, balstoties uz programmas ārējo uzvedību. Ārējo uzvedību nosaka viens no diviem avotiem:

- prasību specifikācija — ir zināms, kā programmai ir jāstrādā;
- pati izpildāmā programma — ir zināms kā programma strādā.

Otrajā variantā testpiemērus var ģenerēt tikai dinamiski. Pirmais variants šajā ziņā ir atvērts plašākam testpiemēru ģenerēšanas metožu klāstam. Pie tam otrajā variantā ir iespējama tikai

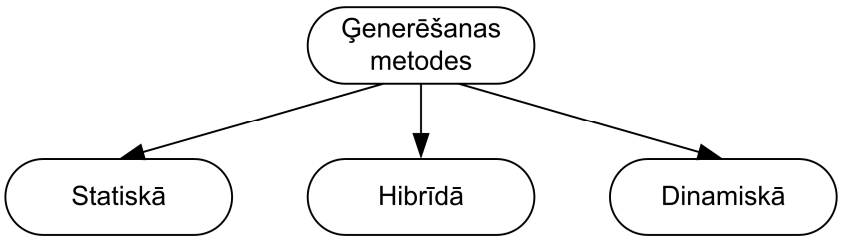
pētnieciskā testēšana (*exploratory testing*), kad programma sākumā tiek darbināta ar gadījuma ievaddatiem, un tikai gūstot kādus rezultātus, ģenerators var veikt secinājumus par tās darbību un modelēt to. Pirmajā variantā ir iespējama gan pētnieciskā testēšana, gan uz specifikācijām balstītas metodes (*specification-based testing*). Katrā gadījumā pirmais variants ir daudz piemērotāks automātiskai testpiemēru ģenerēšanai.

Pelēkās kastes pieeja zināmā mērā kombinē baltās un melnās kastes pieejas. Parasti tas nozīmē, ka pirmkodu izmanto testpiemēru projektēšanai, taču izpilda tos testus, uzskatot programmu par melno kasti, t.i., no ārējā, funkcionālā viedokļa.

Tā kā testpiemēru ģenerēšanas uzdevums ir ierobežots ar testpiemēru projektēšanas automatizāciju, šāda pieeja reducējas uz baltās kastes testpiemēru ģenerēšanas metodēm. Testpiemēru ģenerēšanas kontekstā par pelēko kasti būtu lietderīgāk saukt tādu programmu, kurai ir pieejama tikai daļa no pirmkoda. Šādas situācijas rodas, kad programma izmanto kādas ārējās bibliotēkas vai ietvarus. Tie izmantotie komponenti šajā ainā ir melnās kastes.

Testpiemēru ģenerēšanas metožu klasifikācija pēc darbināšanas veida

Pēc darbināšanas scenārija ģenerēšanas metodes var iedalīt statiskajā, dinamiskajā un hibrīdajā ģenerēšanā (1.3. att).



1.3. att. Klasifikācija pēc darbināšanas veida

Statiskā testpiemēru ģenerēšana analizē programmu statiski, izmantojot vienīgi simboliskās izpildes metodes ievaddatu noteikšanai, kuru rezultātā programma izies noteiktus zarus vai ceļus. Tas tiek darīts reāli nedarbinot programmu [46, 73]. Pamatideja ir izpētīt visus iespējamus darbināšanas ceļus, ņemot vērā visas iespējamas mainīgo izmaiņas programmas darbināšanas gaitā.

Kaut arī statiskā testpiemēru ģenerēšana tiek aktīvi pētīta jau vairāk kā 30 gadus [73], tā joprojām var apstrādāt tikai visvienkāršākās programmas. Kaut cik nopietnām programmām

tā nav lietojama, jo simboliskā spriešana nav iespējama ļoti plašam programmu izteiksmju veidu klāstam, piemēram, tādām kā:

- darbības ar rādītājiem;
- vairākas aritmētiskas operācijas;
- sistēmas funkciju izsaukumi;
- bibliotēku funkciju izsaukumi;
- un vairākas citas.

Dinamiskā ģenerēšana darbina programmu, parasti sākot ar gadījuma izvēlētiem ievaddatiem, un darbināšanas laikā savāc informāciju par ievaddatu ierobežojumiem un zaru predikātiem. Tad ierobežojumu analizētājs ģenerē novirzes iepriekšējos ievaddatos, lai nākamā programmas darbināšana ietu pa alternatīvo zaru. Šis process tiek turpināts kamēr netiks sasniegts programmas beigu stāvoklis vai netiks izpildīts noteikts programmas ceļš [46, 74].

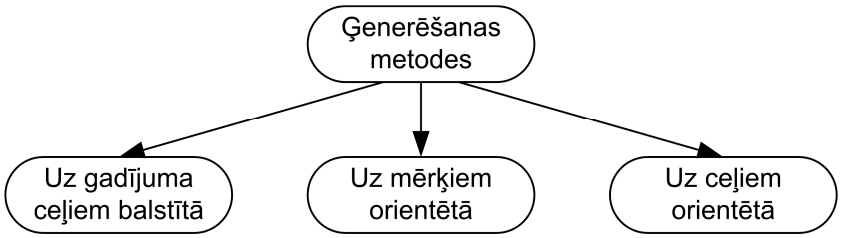
Dinamisko testpiemēru ģenerēšanu var uzskatīt par statiskās ģenerēšanas paplašinājumu ar papildus izpildes laika informāciju, pateicoties kurai ģenerēšana kļūst daudz jaudīgākā un plašāk piemērojama. Vispār var lietot parasto simboliskās izpildes dzini, papildinot to ar konkrētām papildus vērtībām, ko ierobežojumu analizētājs nevar pats izsecināt.

Hibrīdā ģenerēšana apvieno abu iepriekšējo pieeju iespējas [48]. No vienas puses, tā dara visu iespējamo, lai ģenerētu testpiemērus statiski. No otrās puses tajos gadījumos, kad tas izrādās neiespējams vai pārāk sarežģīts, tā izmanto dinamisko programmas darbināšanu, lai iegūtu papildus informāciju.

Programmas darbināšana varētu būt izpildes laika un resursu ziņā dārgs process, līdz ar to hibrīdā pieeja cenšas pēc iespējas minimizēt reālās darbināšanas skaitu. Rezultātā priekšrocība ir tāda, ka testpiemēru ģenerēšana norīt ātrāk. Galvenais hibrīdās pieejas trūkums ir tāds, ka ģenerēšanas algoritmi ir ievērojami sarežģītāki, nekā tīri statiskajai vai dinamiskajai pieejai, jo spriešana par to, kuros gadījumos ir jāpiemēro simboliskā darbināšana, un kuros — dinamiskā, parasti nav triviāla.

Testpiemēru ģenerēšanas metožu klasifikācija pēc ceļu veidiem

Ferguson un Korel [38] piedāvā testu ģenerēšanas klasifikāciju pēc ceļu izvēles veida trīs paveidos (1.4. att.).



1.4. att. Klasifikācija pēc ceļu veidiem

Uz gadījuma ceļiem balstīta ģenerēšana ir vienkāršākais testpiemēru ģenerēšanas paveids. Ceļš, kuram tiks izvēlēti ievaddati, var būt jebkurš. Vienīgais ierobežojums — ceļam ir jābūt izpildāmam, t.i., programmai ir jāizpildās no sākuma stāvokļa līdz beigu stāvoklim.

Šīs pieejas priekšrocība ir tāda, ka ievaddati var būt gadījuma ģenerēti. Ģenerētiem ievaddatiem ir jāapmierina noteikti priekšnosacījumi. Ceļu selektoram ir jāanalizē programmas pirmkods vai modelis, lai noteiktu šos priekšnosacījumus. Kad tie ir zināmi, tiem atbilstošo ievaddatu ģenerēšana ir triviāls uzdevums, jo ceļu selektoram faktiski nav jāveic ceļu izvēle. Šī pieeja ir pielietojama praktiski jebkurai programmai, kurai var noteikt ievaddatu priekšnosacījumus, neatkarīgi no tā, vai pirmkods ir vai nav pieejams. Algoritma realizācija ir triviāla. Līdz ar to šo metodi bieži izmanto etalonuzdevumos (*benchmarks*) citu testpiemēru ģenerēšanas metožu efektivitātes novērtēšanai. Sarežģītākām metodēm noteikti ir jābūt efektīvākām par gadījuma ceļu metodi (savādāk, no tās metodes nav praktiskā labuma), un pēc noteiktiem kritērijiem šo efektivitāti var skaitliski novērtēt.

Galvenais šīs pieejas trūkums ir tāds, ka ir grūti sasniegt vēlamu pārklājumu [34]. Daži ceļi, zari vai pat stāvokļi var palikt nepārbaudīti.

Uz mērķiem orientētā ģenerēšana (*goal-oriented test case generation*) ir nedaudz labāka pieeja. Testpiemēru ģenerators tajā vairs nevar patvaļīgi izvēlēties korektus ievaddatus, bet tam ir zināmi ierobežojumi uz ceļa īpašībām. Šis ierobežojums ir saucams par mērķi. Konkrēts piemērs šādam mērķim ir noteikta apakšceļa (t.s. nespecifiskā ceļa) pārbaude [34].

Uz ceļiem orientētā testpiemēru ģenerēšana (*path-oriented test case generation*) ir stingrāka no visām trim pieejām. Tā neļauj testpiemēru ģeneratoram brīvi izvēlēties kādu no atbilstošo specifisku ceļu kopas, bet dod tam vienu konkrētu specifisku ceļu (pilnu ceļu no programmas vai apakšprogrammas sākuma līdz beigām), kuram tad arī ir jāatrod attiecīgie ievaddati [34]. No šī viedokļa pieeja ir līdzīga iepriekšējai (faktiski, tas ir iepriekšējās metodes stingrākais speciālgadījums), atšķirība ir tikai tāda, ka nespecifiska ceļa vietā testpiemēru ģeneratoram tiek dots specifisks ceļš.

Metodes priekšrocība ir tāda, ka ar to var nodrošināt ceļu pārklājumu, kas ir labāks par virsotņu vai zaru pārklājumiem. No otrās puses testpiemēru ģeneratora uzdevums kļūst sarežģītāks — atrast ievaddatus, kas atbilst konkrētam specifiskajam ceļam, vai arī secināt, ka tādi ievaddati neeksistē (t.i., specifiskais ceļš nav izpildāms), ir grūtāk.

Testpiemēru ģenerēšanas metožu klasifikācija pēc modeļu paveidiem

Uz modeļiem balstīta testpiemēru ģenerēšana ir plaši izpētīta metožu kopa. Faktiski, gan uz specifikācijām balstītā melnās kastes pieeja, gan arī uz pirmkodu balstītā baltās kastes pieeja var tikt reducētas uz modeļu balstītu pieeju. Kā jau tika atzīmēts baltās kastes pieejā pirmkoda analizētājs transformē pirmkodu kādā vadības vai datu plūsmas grafā. Tas savukārt var atbilst iezīmēto pāreju sistēmai (*LTS, labelled transition system*) vai galīgajai stāvokļu mašīnai (*FSM, finite state machine*). Kas attiecās uz melnās kastes testpiemēru ģenerēšanu, sākotnēja specifikācija var tikt transformēta ļoti daudzos modeļu veidos. Piemērotu modeļu izvēle ir atkarīga gan no programmas īpatnībām, gan no specifikācijas īpatnībām, gan no testēšanas mērķiem. B. Aichernig piedāvā uz modeļiem balstītu testpiemēru ģenerēšanas metožu klasifikāciju atkarībā no modeļu veidiem [4].

Kontraktu veida specifikācijas (*contract-like specifications*) kļuva aktuālas ar projektēšanas pēc kontrakta (*design-by-contract*) pieejas parādīšanos, taču šī koncepcija ir zināma vēl no 70. gadu sākuma. To kopīga raksturiezīme ir tāda, ka tiek precīzi formulēts stāvoklis, kas nosaka operāciju sagaidāmo vidi, kā arī sagaidāmais operācijas darbības rezultāts. Attiecīgas valodas ir zināmas kā uz stāvokļiem balstītas specifikāciju valodas.

Formāli API jeb lietojumprogrammas saskarņu (*Application Programming Interface*) kontrakti ir predikāti mainīgo kopā. Balsoties uz pirmsnosacījumu un pēcnosacījumu predikātiem un izmantojot ierobežojumu analīzes metodes, tiek sameklētas ekvivalenču klases, no kurām tad tiek izvēlēti konkrēti ievaddati testpiemēriem. Tas atšķiras, piemēram, no FSM pieejas, kur attiecīgi pirmsnosacījumi un pēcnosacījumi ir nevis uzdoti, bet tiek izrēķināti pēc nepieciešamības no pāreju ierobežojumiem.

Kontraktu veida specifikāciju pieeja ir realizēta, piemēram Eiffel programmēšanas valodā [85], kuras sintakse ļauj specificēt pirmsnosacījumus un pēcnosacījumus pie operāciju definēšanas. Java valodai var tikt izmantota JML valoda (*Java Modelling Language*) [78], kas šo informāciju ļauj formāli aprakstīt Java komentāros, izmantojot speciālo sintaksi. Eksistē vairāki rīki, kas spēj analizēt JML definētus ierobežojumus gan izpildes laika pārbaudēm, gan arī testpiemēru ģenerēšanai. UML modeļos papildus ierobežojumus var uzdot, izmantojot

OCL (*Object Constraint Language*) valodu [26]. Ja pirmie divi piemēri dod iespēju ģenerēt testpiemērus ar baltās kastes metodi, tad OCL var tikt izmantota melnās kastes testpiemēru ģenerēšanai.

Abstraktie datu tipi jeb ADT (*Abstract Data Types*) ir algebriskas specifikācijas, kas specificē attiecības starp tipa operācijām. Šajās specifikācijās nekas netiek teikts par definējamo objektu iekšējo uzbūvi. ADT sastāv no:

- signatūras, kas deklarē datu tipus un pieejamas operācijas;
- pozitīvo nosacījumu aksiomām.

Šādas aksiomas loģiski precīzi formulē operāciju jēgu un to savstarpējas attiecības. Ir izstrādātas vairākas valodas, kas nodrošina šādu aksiomu pierakstīšanu, piemēram OBJ [42], CASL [89] vai ELOTOS [63]. Galvenā ideja ir šo aksiomu interpretēšana un tādas ievaddatu kopas ģenerēšana, ar kuru palīdzību varētu pārbaudīt aksiomu izpildīšanos pie dažādām loģiskām vērtībām.

Iezīmēto pāreju sistēmas jeb LTS (*Labelled Transition System*) var definēt kā kortežu $LTS = \{S, s_0, \Sigma, \delta\}$, kur S ir stāvokļu kopa, $s_0 \in S$ ir sākuma stāvoklis, Σ ir iezīmju alfabēts, bet $\delta: S \times \Sigma \to S$ ir stāvokļu pāreju attiecība.

Galvenā LTS atšķirība no FSM ir tāda, ka LTS būs bezgalīgs stāvokļu skaits. Valodas, ar kurām var specificēt LTS, ir, piemēram, LOTOS [63], SDL [64] vai Promela [50].

No LTS tika atvasināti vairāki citi modeļu paveidi. Piemēram, IOSTS (*Input/Output Symbolic Transition System*) paplašina LTS ar to, ka stāvoklis satur „atrašanas vietu", kā arī mainīgo un parametru piešķires. Pārejas atbilst ievadei, izvadei vai iekšējai darbībai, pie tam pie pārejas var būt ierobežojumu nosacījumi.

Galīgās stāvokļu mašīnas jeb FSM (*Finite State Machine*) ir ļoti līdzīgas LTS, taču stāvokļu kopa ir galīga, un ir divi alfabēti — ieejas un izejas. Papildus var tikt definēta „uzvedības funkcija", kas pārim „stāvoklis – ieejas simbols" piekārto izejas simbolu.

Populārs FSM attēlošanas veids ir UML stāvokļu pāreju diagrammas. Taču UML piedāvātas iespējas ir ievērojami plašākas, nekā tīriem FSM.

Vēl var minēt tādus modeļu paveidus kā datu plūsmu modeļi [104], Kripke struktūras [51], Petri tīkli [23] utt. Katrs modeļa veids piedāvā noteiktu reālas programmas abstrakciju, kas nodrošina programmas novērošanu no noteikta aspekta.

Ģenerēšanas algoritmi lielā mērā ir atkarīgi no modelēšanas veida. Iespējams, ka tieši ar to ir izskaidrojams fakts, ka testpiemēru automātiska ģenerēšana joprojām netiek plaši lietota programmatūras izstrādes industrijā — katrs atsevišķs modelis vai algoritms piedāvā tikai

vienpusīgu, visai ierobežotu skatu uz testējamo programmu, bet plašākam aptvērumam ir jāizmanto vairāki modeļi un ģenerēšanas algoritmi, kas prasa pārāk daudz darba modelēšanai un daudz zināšanu algoritmu lietošanai.

1.3. Testu izpildes automatizācijas rīku klasifikācija

1.3.1. Klasifikācijas struktūra

Mūsdienās testēšanas atbalstam vispār un sevišķi testu automatizācijai ir pieejami daudzi rīki, kuru skaits ir mērāms simtos. Lai orientētos šajā dažādībā ir nepieciešams pamats, kas ļautu organizēt par rīkiem pieejamu informāciju vienotā un saprotamā sistēmā. Taisnākais šādas organizēšanas ceļš ir klasifikācija, kuras rezultātā rīki tiek sadalīti grupās pēc izraudzītām īpašībām jeb klasifikācijas kritērijiem.

Klasifikācijas kritēriju izvēle ir grūts uzdevums, jo testu automatizācijas rīku īpašības ir daudzas, katram atsevišķi ņemtam rīkam ir arī savas unikālas īpašības, kas piemīt tikai tam. Līdz ar to klasifikāciju ir iespējams veikt dažādos veidos atkarībā no tiem uzdevumiem, ko šai klasifikācijai ir jārisina.

Par klasifikācijas mērķiem tika izraudzīti šādi.

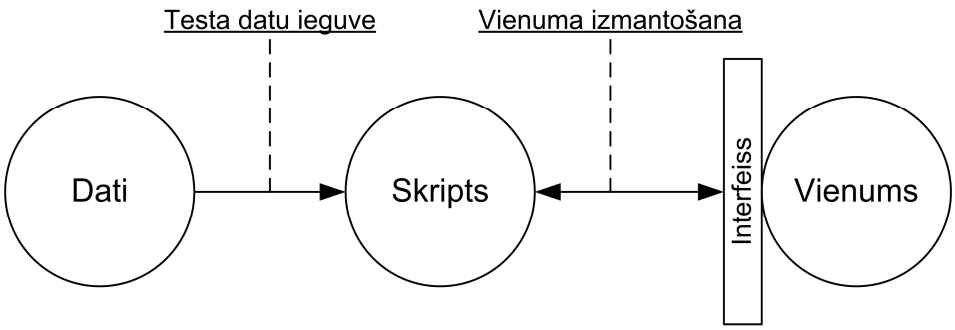
- Klasificēšanas ērtums. Ir nepieciešams rīkus sagrupēt tādā veidā, lai katra atsevišķi ņemta rīka piederību klasei būtu iespējams viegli novērtēt.
- Klasificēšanas viennozīmīgums. Pēc iespējas ir jāizrauga tāds klasificēšanas veids, lai rīku varētu attiecināt uz vienu un tikai vienu klasi.
- Klasificēšanas daudzpusība. Klasificējot, jāņem vērā vairākas būtiskas rīku īpašības, kas varētu raksturot tos no vairākiem būtiskiem aspektiem.
- Klasificēšanas lietderība. Šādai klasifikācijai ir jākalpo arī praktiskiem mērķiem, starp kuriem var atsevišķi izdalīt:
  - rīku izvēles atbalsts. Praktiskajā situācijā, kad ir jāizvēlas testu automatizācijas rīks, un ir zināmi mērķi, kurus vajag sasniegt, klasifikācijai ir jāpalīdz sašaurināt kandidātu rīku loku, lai izvēle būtu veicama no nelielas piemērotāko rīku kopas;
  - testējamības projektēšanas atbalsts. Projektējot programmatūru, ir jādomā par to, kā tā tiks testēta. Testu automatizācijas rīku klasifikācijai jāpalīdz projektētājam pieņemt tādus lēmumus, kas turpmāk vienkāršos testu automatizācijas veikšanu.

Dinamiskā automatizētā testa modelis

Atšķirībā no statiskās testēšanas, kas neparedz programmatūras izpildi, dinamiskajās metodēs programmatūra tiek izpildīta, lai novērtētu tās īpašības [47]. Klasiskās dinamiskās testēšanas gadījumā šo izpildi veic testētājs, iespējams lietojot palīgrīkus (piemēram, kāda atsevišķa moduļa testēšanai varētu izmantot lietotāja saskarni, kas tika izstrādāta tieši šī modeļa testēšanas nolūkā). Šajā pieejā cilvēks ir testa izpildītājs. Automatizēto testu atšķirība ir tāda, ka tests izpildās automātiski, bet cilvēka līdzdalība ierobežojas ar testa (vai vairāku testu kopas) palaišanu un rezultātu analīzi. Taču, lai tas būtu iespējams, testu sākumā vajag automatizēt, t.i., izstrādāt testa programmatūru (*testware*) [39].

Testa programmatūru var iedalīt testa skriptā un testa datos. Skripts ir testa procedūras realizācija, kas ietver darbības ar testējamo vienumu, reakcijas korektuma pārbaudes un citas darbības, kas ir nepieciešamas, lai skripts izpildes laikā varētu novērtēt noteiktas testējamā vienuma īpašības. Testa dati ir dati, ko izmanto skripts, lai izpildītu noteiktus testpiemērus. Testa datus var iedalīt ievaddatos, sagaidāmo rezultātu datos un palīgdatos.

Ņemot vērā minētos aspektus, var uzbūvēt automatizētā testa modeli, kas ir atspoguļots 1.5. attēlā.



1.5. att. Automatizētā testa modelis

Testējamais vienums var būt atsevišķs koda fragments, modulis vai vesela sistēma. Skripts darbina vienumu caur tā interfeisu. Skriptam nav zināma vienuma iekšēja struktūra (ar to testu automatizācijas rīki atšķiras no koda analīzes rīkiem), bet vienīgi tā vienuma interfeiss, caur kuru arī notiek sadarbība — darbību veikšana un korektuma pārbaudes. Skripta sadarbība ar testējamā vienuma interfeisu 1.5. attēlā ir parādīta kā divvirzienu bulta, kas simbolizē to, ka datu plūsma notiek abos virzienos.

Datus skripts iegūst no kāda datu avota, kas ir ilustrēts ar vienvirziena bultu, jo skripts iegūst iepriekš sagatavotos datus, bet pats tos nemaina.

Šajā modelī nav atspoguļoti vairāki aspekti, kas testu automatizācijas rīkiem ir, taču nav tik būtiski šī rīka piemērotības novērtēšanai, piemēram, skripta sadarbība ar testētāju (kā testētājs palaiž testu), vai rezultātu prezentēšana (kā skripts prezentē testa rezultātus pēc izpildes). Šie aspekti drīzāk ir saistīti ar rīka lietošanas ērtumu, kas ir atkarīgs arī no tā lietotāju pieredzes, gaumēm un citiem subjektīviem faktoriem.

Uzbūvētajā modelī ir izdalāmi četri aspekti, kas arī kalpos par klasifikācijas kritērijiem [130].

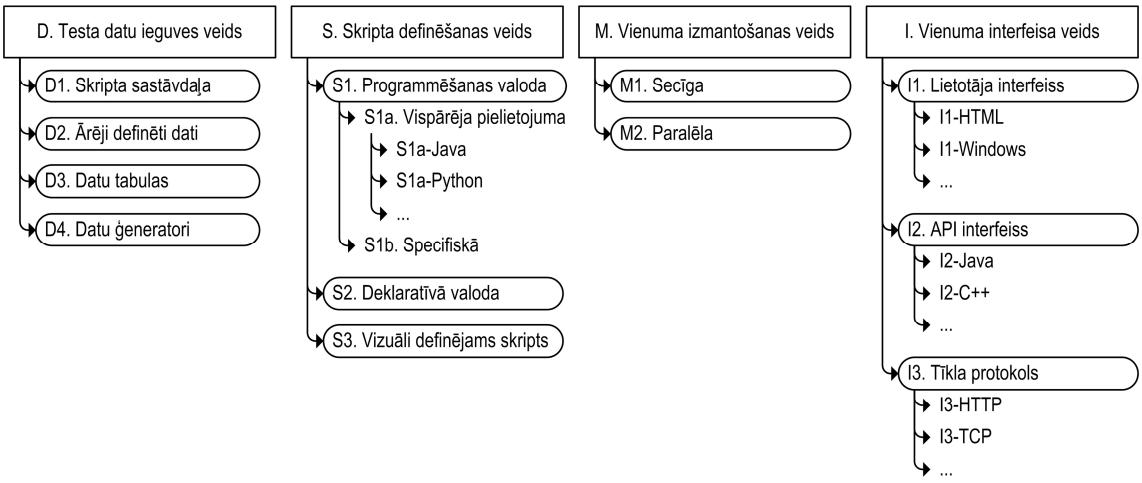
1. Testa datu ieguves veids. Šis kritērijs nosaka, kādā veidā skripts iegūst datus, vai dati ir atdalīti no skripta, vai ir skripta sastāvdaļa. Šis kritērijs tiks apzīmēts ar burtu D.
2. Skripta definēšanas veids. Nosaka, kādā veidā skripts tiek izstrādāts, kāda ir tā struktūra un kā tas tiek interpretēts. Šis kritērijs tiks apzīmēts ar burtu S.
3. Vienuma izmantošanas veids. Nosaka veidu, kādā izmanto vienumu, vai darbības tiek veiktas secīgi vai paralēli. Šis kritērijs tiks apzīmēts ar burtu M.
4. Vienuma interfeisa veids. Nosaka, ar kādu sistēmas līmeni sadarbojas skripts. Šis kritērijs tiks apzīmēts ar burtu I.

Citi kritēriju piemēri, ko varētu atvasināt no šī modeļa, piemēram, tādi kā datu definēšanas veids un vienuma veids, nav būtiski priekš testu automatizācijas rīku piemērotības analīzes. Dati var glabāties jebkurā veidā, rīkam ir svarīgs tikai tas, kā skripts iegūs šos datus, bet šis aspekts atbilst pirmajam kritērijam no saraksta. Vienuma veids, t.i., tā uzbūves daba arī nav būtiska no rīka viedokļa, jo rīks darbojas ar vienumu tikai caur interfeisu. Kaut arī interfeisa tips var būt atkarīgs no paša vienuma tipa, tieši interfeiss ir tas, kas ir svarīgs skripta darbināšanai.

1.3.2. Klasifikācijas kritēriji un rīki

Klasifikācijai tika izvēlēti četri kritēriji, pamatojoties uz 1.3.1. sadaļā aprakstīto modeli. Katrs kritērijs ietver vairākus variantus (klases), kurus apzīmēsim ar atbilstoša kritērija apzīmējuma burtu un klases numuru, piemēram, D3, kur D ir kritērija apzīmējums, un 3 — apzīmē trešo klasi pēc kritērija D.

Visus kritērijus ar iespējamām vērtībām apkopo 1.6. attēls.



1.6. att. Rīku klasifikācijas kritēriji

Uz piedāvātās klasifikācijas pamata tika analizēti un novērtēti 32 rīki, veidojot to rīku iedalījumu klasēs pēc katra kritērija. Šajā sadaļā tiks sīkāk apskatīti izraudzītie kritēriji, katram kritērijam tiks norādīts, kādās klasēs pēc tā var sadalīt testu automatizācijas rīku telpu, kā arī rīki, kas atbilst katrai klasei pēc noteiktā kritērija.

D — Testa datu ieguves veids

Testa dati ir svarīgs testpiemēra aspekts. Datu domēnu lielums ir tāds, ka pilna testēšana nav praktiski iespējama [47]. Automatizācija šeit palīdz tikai daļēji, jo kaut arī testu automātiskā izpilde ir daudz ātrāka, salīdzinot ar manuālo, pat dažu tūkstošu testu izpilde var aizņemt ievērojamu laiku, it īpaši ja testējamā vienuma darbības ātrums nav liels (kā parasti ir, ja testējamais vienums ir vesela sistēma). Tāpēc projektējot testpiemēru ir nepieciešams izvēlēties „labākos“ datu komplektus, t.i., tādus komplektus, kas atklās defektu ar lielāku varbūtību.

No testu automatizācijas rīka darbības viedokļa nav svarīgi, kādā veidā dati tiek iegūti vai kādā veidā tie tiek glabāti. Galvenais ir tas, kā skripts darbināšanas laikā iegūs šos datus. Šeit ir iespējami četri varianti (klases), kas aprakstīti to universāluma augošā secībā. Katram datu ieguves variantam atbilst sava rīku klase. Daži rīki var atbalstīt vairākus no šiem datu iegūšanas veidiem. Tad uzskatīsim, ka rīks pieder klasei universālākai no kandidātu klasēm, t.i., klasei ar lielāku numuru.

- D1 — Dati ir skripta neatņemama sastāvdaļa. Tas nozīmē, ka skripts satur datus kā konstantas vērtības un izpildās vienmēr ar vienu un to pašu datu komplektu.

- D2 — Ārēji definēti dati. Dati var glabāties atsevišķi no skripta un citā formā. Datu komplekts šajā variantā joprojām ir viens, taču to ir iespējams mainīt nemainot pašu skriptu.
- D3 — Datu tabulas. Tabulā var glabāties vairāki testa datu komplekti vienam skriptam. Skripts tādā gadījumā var izpildīties tik daudz reizes, cik tabulā ir datu komplektu. Šādi tiek panākta vairāku testpiemēru izpilde, kuriem atbilst viena un tā pati testa procedūra.
- D4 — Datu ģeneratori. Datu ģeneratori ir atsevišķi komponenti, kas pēc noteiktiem likumiem automātiski ģenerē testa datus. Skripts šajā variantā arī darbojas iteratīvi, katru reizi izpildot jaunu testpiemēru, taču testētājs tā vietā, lai pašam noteiktu datu vērtības, uzdod likumus, kādiem šiem datiem ir jāatbilst.

D1 klasē dati nav atdalīti no skripta. Tas nozīmē, ka ja ir nepieciešams datus nomainīt ar citiem, ir jāmodificē pats skripts.

Šai pieejai galvenā priekšrocība ir testu automatizācijas vienkāršums. Testu automatizētājam nav jārūpējas par to, kā iegūt datus, jo dati ir atrodami tajā pašā vietā, kur tie tiek izmantoti. Skripta izpilde arī mēdz būt ātrāka, jo automatizācijas rīkam nav jāizpilda liekas darbības datu iegūšanai.

Datu statiskums sarežģī automatizēto testu uzturēšanu. Testa datu izmaiņu veikšanai ir nepieciešams modificēt pašu skriptu. Testētājam (atšķirībā no testu automatizētāja) var būt grūti orientēties testa skriptā, it īpaši ja tas ir uzdodams koda veidā, bet testētājam nav vai ir maz programmēšanas pieredzes. Tāpēc ir risks, ka testētājs izdarīs kaut ko ne tā un sabojās skripta loģiku.

D2 rīku klase dati ir atdalīti no skripta un tos var mainīt, nemodificējot pašu skriptu. Datus šajā gadījumā var uzskatīt par skripta izpildes parametriem. Var būt arī iespēja izpildīt vienu un to pašu skriptu ar vairākiem datu komplektiem. Atšķirībā no D1 klases, kur šāda rīcība prasītu arī skripta kopēšanu, D2 klases rīkiem tas nozīmētu skripta izsaukumu kopēšanu, pašam skriptam paliekot nemainītam.

Šajā pieejā ir ērti atdalīt testētāja un testu automatizētāja pienākumus. Testu automatizētājs ir atbildīgs par skripta sagatavošanu viņam ērtajā formā (piemēram, kods kādā programmēšanas valodā), bet testētājs ir atbildīgs par testpiemēra identificēšanu un atbilstošo datu sagatavošanu viņam ērtajā formā (piemēram, lietojot vizuālo saskarni vai kādu vienkārša formāta teksta failu). Datu izmaiņas ir iespējams veikt nemainot pašu skriptu, un arī skriptu ir iespējams modificēt nepieskaroties sagatavotajiem datiem.

Šai gadījumā skriptu sagatavošanas uzdevums ir sarežģītāks, jo testu automatizētājam ir jārūpējas par to, kā iegūt datus no ārēja avota. Rīks var nodrošināt, lai šis uzdevums varētu

tikt izpildīts maksimāli vienkārši, taču vienalga papildus darbs ir nepieciešams. No otras puses, testētājs, kas sagatavo datus, ir ierobežots ar viena testu komplekta izmantošanu vienam skriptam, vai, labākajā gadījumā, tam ir jāveic papildus darbs skripta izsaukumu kopēšanai, kas var būt sarežģītāks vai vienkāršāks uzdevums atkarībā no rīka īpašībām.

D3 klases rīki spēj nodrošināt viena skripta izpildi ar dažādiem datiem, kuri ir definēti ārējā datu tabulā. Vienā tabulā tiek definēti vairāki datu komplekti, kas atbilst dažādiem testpiemēriem, bet skripts ir atbildīgs par šiem testpiemēriem atbilstošas testa procedūras izpildi. Līdzīgi kā D2 klases rīkiem testa datus var uzskatīt par skripta izpildes parametriem, taču tagad vairāki parametru komplekti tiek definēti vienā labi pārskatāmā struktūrā. Parasti tabulas kolonnas atbilst dažādiem skripta parametriem, bet tabulas rindas — dažādiem testpiemēriem. Pašas tabulas var glabāties Excel failos, CSV failos vai pat relāciju datubāzēs.

Šajā gadījumā dati ir reāli atdalīti no skripta un vienam skriptam var atbilst vairāki datu komplekti. Tas nozīmē ievērojamu darbietilpības ekonomiju, jo jauna testpiemēra pievienošana testu komplektam, ja priekš attiecīgas testa procedūras tests jau eksistē, nozīmē tikai jaunas rindiņas (vai kolonnas) pievienošanu datu tabulai.

Tabulās organizēto informāciju ir viegli uztvert gan testētājiem, gan testu automatizētājiem, tāpēc tās droši var kalpot par sava veida saskarni starp tiem, kas datus definē, un tiem, kas tos datus izmanto.

Vienu datu tabulu potenciāli var atkārtoti izmantot vairākos testos, piemēram, pasūtījumu tabulu varētu izmantot gan pasūtījumu veikšanas testēšanai, gan pasūtījumu apstrādes testēšanai, gan pasūtījumu dzēšanas testēšanai. Tas nozīmē, ka ne tikai vienam skriptam var atbilst vairāki datu komplekti, bet arī vienai datu komplektu tabulai var atbilst vairāki skripti, kas to izmanto.

Skriptu uzbūve šīs klases rīkiem ir sarežģītāka, salīdzinot ar D1 klases rīkiem, jo ir jānodrošina datu ieguve no ārēja avota, taču ir nesarežģītāka, kā D2 klases rīkiem, kuru skriptiem ir tāda pati īpašība.

Vēl viens D3 rīku klases trūkums ir tas, ka dažādus testpiemērus, kas ir definēti vienā datu tabulā, parasti neatdala vienu no otra, t.i., rīki to uztver kā vienu testu, kura izpilde rezultātā dod vienu izpildes rezultātu, pozitīvu vai negatīvu. Lai secinātu, kura konkrēta testpiemēra dēļ viss testa rezultāts ir negatīvs, var būt nepieciešama papildus analīze, kas var prasīt ne tikai testētāja, bet arī testu automatizētāja kompetences. Taču šis trūkums ir galvenokārt saistīts ar esošo rīku realizācijas īpatnībām, nevis pašas D3 pieejas īpatnībām.

Vairākums komerciālo automatizācijas rīku, kas ir paredzēti lietotāja saskarnes testu automatizācijai, pieder tieši šai klasei. Tādi rīki kā QuickTest Professional [54], TestPartner [86, 92], TestComplete [11] atbalsta šo pieeju.

D4 klasē varētu pieskaitīt rīkus, kas izmanto nevis konkrētus datus, bet kādā veidā definētos noteikumus, no kuriem testa dati tiek ģenerēti. Rīki, kas atbalsta šo pieeju šobrīd nav plaši izplatīti un nav arī pietiekami attīstīti, lai nodrošinātu kvalitatīvu un uzticamu darbu. Līdz ar to D4 klasi varētu arī atsevišķi neizdalīt, taču pēc autora domām, šāda pieeja nākotnē varētu attīstīties un kļūt populāra, tāpēc to ir vērts apskatīt.

Šāda pieeja var būt visai noderīga gadījumos, kad ir jāizpilda daudz testpiemēru, kuru dati ir viegli ģenerējami, vai gadījumos, kad izmantojamie dati ir atkarīgi no vides, kura būtu jāanalizē pirms testu uzsākšanas. Tad datu ģenerēšana var būt izdevīgāka par to manuālo definēšanu.

Datu ģeneratora konfigurēšana (vai programmēšana) ir sarežģīts uzdevums, tāpēc cilvēki, kas parasti atbild par testa datu sagatavošanu, parasti nav spējīgi to veikt patstāvīgi, viņiem ir nepieciešama programmētāju, administratoru vai rīka ekspertu palīdzība. Pie tam tiek arī samazināta testu uzticamība, jo ģenerējamo datu kvalitāte ir grūti novērtējama.

Izplatītākie testu automatizācijas rīki šo pieeju neatbalsta. Datu ģenerēšanas funkcionalitāti var nodrošināt, realizējot datu ģeneratoru atsevišķi (piemēram, kā testa skripta apakšprogrammu), vai lietojot ārējos rīkus — testu ģeneratorus. No testu ģeneratoriem izplatīti ir, piemēram, datubāzu testu ģeneratori, tādi kā forSQL [40] vai DataGen [33]. Šiem un arī citiem testu ģeneratoriem pašlaik ir ļoti šaurs pielietojumu loks — tie tiek koncentrēti uz kādas vienas īpašības testēšanas, atšķirībā no citām klasēm, kurām piederošie rīki ir daudz universālāki.

S — Skripta definēšanas veids

Testa skripts ir galvenais testu automatizācijas objekts. Ar skriptu saprot izpildāmo testa procedūras realizāciju, kas nodrošina viena vai vairāku testpiemēru automātisku izpildi. Skripta izstrādes sarežģītība ir atkarīga gan no testējamā vienuma dabas, gan no rīka īpašībām. Rīkus pēc skriptu definēšanas veida var iedalīt šādās klasēs.

- S1 — Programmēšanas valodā definējams skripts. Šajā gadījumā skripta realizācijai tiek izmantota vai nu universāla, vai specializēta programmēšanas valoda.
- S2 — Deklaratīvajā valodā definējams skripts. Šajā gadījumā skripts arī tiek realizēts kā kods teksta formā, tikai atšķirībā no pirmās klases tiek lietota deklaratīva valoda.

- S3 — Vizuāli definējams skripts. Skripts tiek izstrādāts, lietojot tam paredzēto rīka vizuālo lietotāja saskarni.

Neatkarīgi no skripta definēšanas veida, tā izstrāde prasa speciālas zināšanas par rīku un tā iespējām. Tātad jebkurā gadījumā skriptu izstrādātājam tās zināšanas jāapgūst. Bet apgūšanas laiks un nepieciešamas priekšzināšanas ir stipri atkarīgas no konkrētas rīka klases. Tagad izskatīsim minētās klases sīkāk.

S1 rīku klasē, kur skripti ir definējami programmēšanas valodā, ar programmēšanas valodu tiek saprasta jebkura valoda, kas ir pilnīga Tjuringa nozīmē (*Turing complete language*), t.i., valoda atbalsta ne tikai secīgu komandu izpildi, bet arī zarošanos pēc nosacījumiem un ciklus. Papildus tam valoda var atbalstīt procedurālo programmēšanas stilu (koda grupēšanu funkcijās vai procedūrās) vai objektorientēto stilu (klašu veidošanu).

Šo rīku klasi var iedalīt divās apakšklasēs.

- S1a — Rīki, kuros skripti tiek veidoti, lietojot kādu no universālajām valodām, piemēram, Visual Basic, Java, C, vai plaši izmantojamām testēšanai specifiskām valodām, piemēram, TTCN [126]. Šo apakšklasi var tālāk sadalīt grupās pēc izmantotās valodas, piemēram, lietot apzīmējumu S1a-Java, vai S1a-VBA. Rīkus, kas atbalsta skriptu veidošanu vairākās valodās, apzīmēsim S1a-Multiple.
- S1b — Rīki, kuriem ir sava specifiska programmēšanas valoda. Starp piemēriem var minēt WinRunner (TSL valoda) [129] vai OpenSTA (SCL valoda) [95].

Galvenā S1 klases rīku priekšrocība ir skriptu elastīgums, kas ļauj likt skriptam darīt dažādas iepriekš neparedzētas lietas vai apstrādāt nestandarta situācijas. Pateicoties tam, ka vispārīgas pielietošanas valodām parasti ir pieejami brīvi pieejamie moduļi un bibliotēkas, rīka iespējas var paplašināt bez liekām pūlēm. Pat ja nav pieejama vajadzīga bibliotēka, to tomēr pēc nepieciešamības ir iespējams izstrādāt. Un tas neprasa paša rīka modificēšanu.

Šīs klases ievērojamākais trūkums ir tāds, ka testu automatizētājam ir nepieciešamas programmēšanas iemaņas. Tas nozīmē, ka lietojot šos rīkus, vienam cilvēkam ir grūti apvienot testētāja un testu automatizētāja lomas. Katrai no šīm lomām ir nepieciešamas savas specifiskas kompetences, līdz ar to šo lomu atdalīšana kļūst par vienu no faktoriem, kas nosaka testu automatizācijas efektivitāti.

S2 rīku klasē tiek izmantotas deklaratīvās valodas, t.i., valodas, kurās skriptu var aprakstīt, nevis uzprogrammēt. Parasti tās ir paredzētas skriptu komandu stingras secības aprakstam. Tātad šīs klases rīku skripti principā nevar pieņemt lēmumus, veikt iekšējos ciklus utt. Katrs skripts ir noteikta soļu secība, pie tam soļi vienmēr izpildās vienā un tajā pašā secībā.

Un tomēr skripts ir rakstāms kā kods teksta formā. Visbiežāk šim mērķim tiek izmantots XML formāts, kas ir labi piemērots šāda veida informācijas uzturēšanai.

S3 klases rīkiem skriptēšanas valodas tā vārda klasiskajā nozīmē nav. Skripts tiek veidots ar rīka vizuālās lietotāja saskarnes palīdzību. Caur vizuālo saskarni tiek uzstādīti arī konfigurācijas parametri, tātad lietotājam praktiski nav jāstrādā teksta režīmā.

M — Vienuma izmantošanas veids

Veids kā sadarbojas skripts ar testējamo vienumu, ja abstrahēties no skripta dabas un testējamā vienuma interfeisa dabas, nosaka vienīgi to, vai skripts izpilda savas darbības secīgi vai paralēli. Līdz ar to pēc šī kritērija var izdalīt divas klases.

- M1 — Secīga komandu izpilde. Skripts strādā vienā eksemplārā izpildot visus savus soļus (komandas) secīgi vienu aiz otra.
- M2 — Paralēla skriptu izpilde. Rīks spēj palaist vairākus skriptu eksemplārus paralēli, tādējādi imitējot vairākus vienlaicīgus vienuma klientus, ar vārdu „klients“ saprotot vai nu lietotāju, vai arī citus vienumus.

Tātad skriptu eksemplāru iespējamais skaits nosaka šo divu klašu atšķirību. Sākumā var šķist, ka otrās klases rīki ir pārāki par pirmās klases rīkiem, taču patiesībā tā nav, jo paralēlas izpildes nodrošināšana ir saistīta ar būtiskiem trūkumiem, ko aprakstīsim, apskatot tās klases sīkāk.

M1 klasei ir pieskaitāmi rīki, kas izpilda skriptu vienā eksemplārā, tātad tie rīki, ko mēdz saukt par funkcionālās testēšanas rīkiem, kā arī vienībtestēšanas rīki.

Galvenā šīs rīku klases priekšrocība ir M2 klases trūkumu neesamība. Piemēram, funkcionalitātes testēšanu lietotāja saskarnes līmeni var realizēt tikai ar M1 klases rīkiem. M1 klases rīki ir paredzēti testēšanai viena klienta režīmā.

Šādu rīku skriptus var būt grūti adaptēt veiktspējas testēšanas vajadzībām, jo veiktspējas testēšanai ir nepieciešami M2 klases rīki. Kaut arī daži rīku komplekti iekļauj gan M1, gan M2 klases rīkus, lai pārklātu dažādas testēšanas vajadzības, un kaut kādā ziņā atvieglo šo transformācijas procesu, tas vienalga nav viennozīmīgs un prasa laiku un iemaņas, lai to izpildītu.

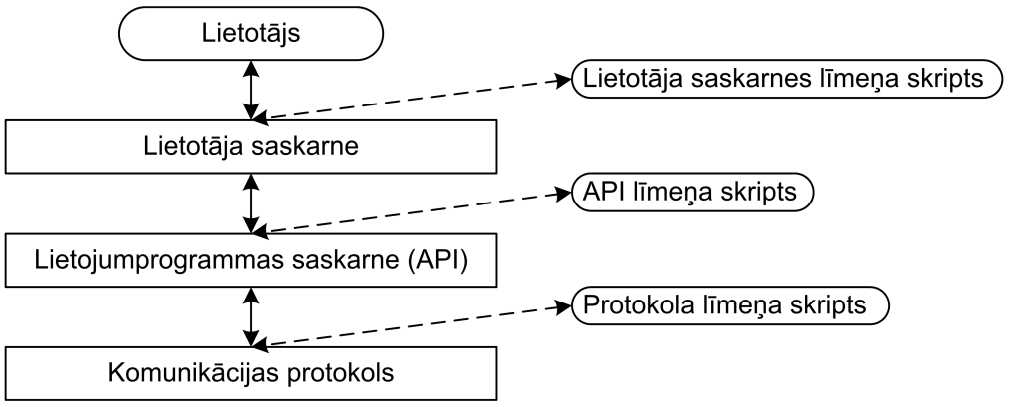
M2 klasē ietilpst rīki, kas ir paredzēti galvenokārt veiktspējas testēšanai. Skriptu eksemplārus tajos var izpildīt paralēli, lai emulētu vairākus vienuma klientus. Vairākus problēmu tipus nav iespējams atklāt viena klienta režīmā. Šāda veida problēmu cēloņi var būt

slikta pavedienu sinhronizācija, datu bloķēšanās vai arī veiktspējas un mērogojamības problēmas. M2 klases rīki ir paredzēti tieši šādu problēmu atklāšanai.

Galvenais trūkums šāda veida rīkiem ir ierobežotas iespējas attiecībā uz atbalstītiem testēšanas līmeņiem (testējamo vienumu interfeisu veidiem). Parasti šādi rīki neatbalsta testēšanu lietotāja saskarnes līmeni, galvenokārt tāpēc, ka vairāku skriptu eksemplāru izpilde prasītu arī vairāku klienta programmatūras eksemplāru izpildi. Klienta programmatūras eksemplārus būtu jāizvieto vienā ekrānā, kas fiziski ir maz iespējams. Līdz ar to M2 klases rīki strādā galvenokārt zemākajos interfeisu līmeņos.

I — Vienuma interfeisa veids

Vienuma interfeisa veida kritērijs nosaka, kādā programmatūras slānī testu automatizācijas rīks veic darbu. Testēšanas uzdevumos ietilpst tādu scenāriju izpilde, kas simulētu kādu situāciju, kura varētu (kaut teorētiski) izveidoties reālas ekspluatācijas laikā. Lai veiktu šādu simulāciju ir nepieciešams aizvietot kādu sistēmas objektu ar citu, kas šo simulāciju veiks. Shematiski tās iespējas ir parādītas 1.7. attēlā.



1.7. att. Skriptu darbības līmeņi

Šāds sadalījums līmeņos testu automatizācijas rīku klasifikācijas mērķiem ir īpaši piemērots, jo ir viegli novērtēt, kādai no klasēm konkrēti rīki pieder. Katras klases rīkiem ir savas specifiskas īpašības, kuras padara testu automatizāciju ērtāku tieši konkrētajā līmenī.

Tātad pēc testējamā vienuma interfeisa veida rīkus var iedalīt trijās klasēs.

- I1 — Lietotāja saskarnes līmenis. Rīks emulē reālo lietotāju, veicot viņam raksturīgas darbības un validējot vizuālo objektu stāvokli dažādos testa izpildes posmos.

- I2 — Lietojumprogrammas saskarnes (API) līmenis. Rīks emulē API klientu, kas reālajā sistēmā varētu būt lietotāja saskarne vai kāds cits sistēmas slānis, vai pat cita sistēma.
- I3 — Komunikācijas protokola līmenis. Šajā gadījumā rīks emulē klientu tīkla komunikācijas nozīmē. Rīks sūta un saņem baitus tā, it kā tas būtu īstais klients.

Šo klasifikācijas kritēriju var nosaukt par visbūtiskāko. Tieši šo kritēriju mēdz uzsvērt, izvēloties testu automatizācijas rīku kādam konkrētam uzdevumam. Tas ir izskaidrojams ar to, ka rīka klases izvēle bieži vien tieši izriet no uzdevuma formulējuma.

I1 rīku klases galvenā atšķirība ir tāda, ka skriptam ir nepieciešams spēt uztvert informāciju, kas ir paredzēta cilvēka uztveršanai, un veikt darbības, kuru veikšana bija paredzēta cilvēkam, kas sēž pie datora. No tā izriet arī galvenā tehniskā sarežģītība, kas ir saistīta gan ar šādu rīku realizāciju, gan ar to izmantošanu — objektu atpazīšanas problēma. Bez objektu atpazīšanas, lietotāja saskarnes līmeņa skripti būtu spiesti strādāt ekrāna koordinašu un tastatūras taustiņu kontekstā, un tāpēc būtu ārkārtīgi jūtīgi pret minimālām izmaiņām saskarnē. Tāpēc mūsdienās šādu rīku praktiski nav to praktiskas nelietderības dēļ, un mēs tos šeit neapskatīsim.

Nepastāv kaut kāda vienota vizuālo objektu programmēšanas standarta. Katrai tehnoloģijai ir savas īpatnības, kā ekrānā tiek zīmētas pogas, ievadlauki, logi utt. Līdz ar to šīs klases rīkus varētu tālāk iedalīt vairākās apakšklasēs pēc konkrētas vizuālo objektu zīmēšanas tehnoloģijas:

- I1-HTML — atbalsta tīmekļa lapu objektus;
- I1-Windows — tie ir standarta Windows objekti;
- I1-Swing — Java standarta bibliotēkas sastāvdaļa, kas ļauj veidot lietotāja saskarnes;
- I1-WinForms — formas, ko zīmē .NET lietojumprogrammas;
- I1-Multiple — līdzīgi kā ar S1a-Multiple, šādi tiks apzīmēti rīki, kuri ir universālāki un atbalsta vairākas vizuālo objektu tehnoloģijas.

I1-HTML apakšklases rīkus var sīkāk iedalīt divās grupās: I1-HTML-DOM, kur objekti tiek uztverti saskaņā ar DOM modeli (šī apakšklase ir tuva I2 klasei, jo darbam DOM modeļa līmenī parasti tiek izmantots API), un I1-HTML-Browser, kas uztver objektus no pārlūkprogrammas tā, kā tie reāli parādās ekrānā.

I2 klases rīki veic testēšanu, izsaucot API funkcijas un tādā veidā imitējot API klientus. Sīkāk šo klasi var iedalīt apakšklases pēc API veida, piemēram:

- I2-Java — rīks izsauc Java funkcijas un validē to atbildi;

- I2-.NET — līdzīgi kā iepriekšējā apakšklasē rīks izsauc .NET funkcijas, t.i., funkcijas, kas ir realizētas kādā no .NET valodām;
- I2-COM — rīks izsauc kādas no COM (*Component Object Model*) metodēm;
- vairāki citi pēc līdzīga principa.

No programmētāju viedokļa API līmeņa testēšanas rīki ir daudz ērtāk lietojami, nekā lietotāja saskarņu līmeņa rīki, jo pazūd visas sarežģītības, kas ir saistītas ar objektu atpazīšanu. Līdz ar to API līmeņa testu izstrāde gandrīz neatšķiras no cita veida programmatūras izstrādes. Pie šīs klases pieder vairākums vienībtestēšanas rīku.

I3 klases rīki strādā komunikācijas protokola līmenī. Darbs komunikācijas protokola līmenī ir raksturīgs slodzes testēšanas rīkiem, jo tieši emulējot komunikāciju starp klienta un servera programmatūru, var noslogot serveri, nepalaižot klienta programmatūru. Tātad ir iespējams ģenerēt pietiekami lielu slodzi, jo resursu patēriņš uz vienu emulējamo klientu būs mazs.

Pēc atbalstāmā protokola tipa šīs klases rīkus var iedalīt vairākās apakšklasēs:

- I3-HTTP — atbalsta komunikācijas testēšanu tīmekļa protokolam un emulē pārlūkprogrammas darbību;
- I3-TCP — strādā zemajā TCP protokola līmenī un ir spējīgs emulēt jebkuru TCP klienta programmatūru;
- I3-Multiple — atbalsta dažādus protokolus;
- vairāki citi, atbilstoši protokolam.

1.3.3. Rīku klasifikācijas kopsavilkums

Iepriekšējā sadaļā tika apskatīti četri klasifikācijas kritēriji un klases, kurās tie iedala visu testu automatizāciju rīku telpu.

1.2. tabulā ir parādīta pilna katra rīka klasifikācija pēc visiem četriem kritērijiem. Katram minētajam rīkam tabulā ir atrodama atsauce, kur var atrast informāciju par šo rīku, kā arī klases pēc katra no četriem kritērijiem: D (datu ieguves veids), S (skripta definēšanas veids), M (vienuma izmantošanas veids) un I (vienuma interfeisa veids).

1.2. tabula

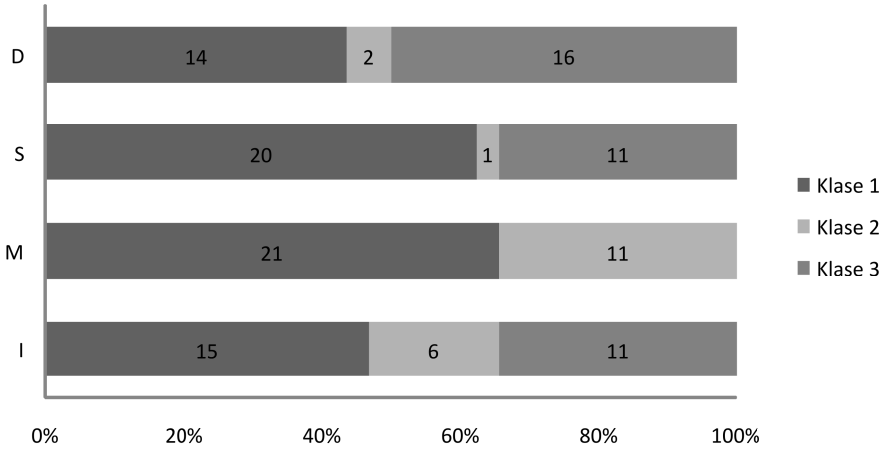
Testu izpildes automatizācijas rīku klasifikācija

Rīka nosaukums	**D**	**S**	**M**	**I**	**Atsauce**
Abbot	D1	S1a-Java	M1	I1-Swing	[2]
Business Process Testing	D2	S3	M1	I1-Multiple	[52]
Canoo WebTest	D1	S2	M1	I1-HTML	[25]
cPAMIE	D1	S1a-Python	M1	I1-HTML	[99]
CppUnit	D1	S1a-C++	M1	I2-C++	[28]
CUnit	D1	S1a-C	M1	I2-C	[29]
HttpUnit	D1	S1a-Java	M1	I3-HTTP	[55]
Jemmy	D1	S1a-Java	M1	I1-Swing	[65]
JMeter	D3	S3	M2	I3-Multiple	[8]
JUnit	D1	S1a-Java	M1	I2-Java	[68]
LoadRunner	D3	S1a-C	M2	I3-Multiple	[53]
TestCast	D3	S1a-TTCN	M2	I3-Mutiple	[35]
NUnit	D1	S1a-Multiple	M1	I2-.NET	[93]
OpenSTA	D3	S1b	M2	I3-HTTP	[95]
OpenTTCN	D3	S1a-TTCN	M2	I3-Multiple	[96]
QALoad	D3	S1a-C++	M2	I3-Multiple	[87]
QF-Test	D1	S3	M1	I1-Swing	[103]
Oracle Functional Testing	D3	S3	M1	I1-HTML	[97]
Oracle Load Testing	D3	S3	M2	I3-HTTP	[98]
QuickTest Professional	D3	S1a-VBS	M1	I1-Multiple	[54]
Rational Functional Tester	D3	S1a-Multiple	M1	I1-Multiple	[58]
Rational Performance Tester	D3	S3	M2	I3-Multiple	[59]
Rational Robot	D1	S1b	M1	I1-Multiple	[60]
Selenium	D1	S3	M1	I1-HTML	[108]
SilkPerformer	D3	S3	M2	I3-Multiple	[21]
SilkTest	D3	S3	M1	I1-Multiple	[22]
TestComplete	D3	S1a-Multiple	M1	I1-Multiple	[11]
TestNG	D1	S1a-Java	M1	I2-Java	[120]
TestPartner	D3	S1a-VBA	M1	I1-Multiple	[86]

1.2. tabulas turpinājums

Rīka nosaukums	D	S	M	I	Atsauce
The Grinder	D1	S1a-Jython	M2	I3-HTTP	[121]
TOSCA	D2	S3	M1	I1-Multiple	[122]
WAPT	D3	S3	M2	I3-HTTP	[125]

Kopējā statistika rīku sadalījumam klasēs ir atspoguļota 1.8. attēlā.



1.8. att. Rīku sadalījums klasēs

Vismazākais rīku skaits ir novērojams klasēs D4 (datu ģeneratori) — neviens rīks, D2 (dati ir atdalīti no skripta) — divi rīki un S2 (deklaratīvajā valodā veidojams skripts) — viens rīks.

D2 un S2 klašu rīku neliels skaits ir izskaidrojams ar to, ka šīs klases varētu uzskatīt par pārejas formām. Datu atdalīšana no skripta ir vienkārši pilnveidojama un pārvēršama par datu tabulu funkcionalitāti (D3 klase), kas paver daudz plašākas testu automatizācijas iespējas. Tas pats attiecās arī uz klasi S2 — ja skripts ir definējams deklaratīvajā valodā, ir relatīvi viegli izveidot vizuālo interfeisu, kas atvieglos skripta izveidi, un rīks pārtop klasē S3.

No analizētajiem rīkiem pie klases D4 (datu ģenerators) nav pieskaitāms neviens rīks. Kaut arī dažiem rīkiem, kuru skripti ir veidojami programmēšanas valodās (S1 klase), pastāv iespēja izstrādāt datu ģenerēšanas algoritmus, bet dažiem pat pastāv iebūvētas funkcijas, kas atvieglo šo uzdevumu, tomēr tās iespējas nav uzskatāmas par rīka pamatfunkcionalitāti. Testa

datu ģenerēšanas rīki pārsvarā pastāv kā atsevišķa rīku klase, kas nenodarbojas ar pašu testu automatizāciju.

1.4. Pirmās nodaļas secinājumi

1.4.1. Veikta testpiemēru automatizētās ģenerēšanas metožu klasifikācija pēc pirmkoda nozīmības, darbināšanas veida, ceļu veidiem un modeļu paveidiem. Klasifikāciju var izmantot piemērotāko testu ģenerēšanas metožu izvēlei atkarībā no uzdevuma īpašībām.

1.4.2. Balstoties uz automatizētā testa modeli, izstrādāta testpiemēru automatizētās izpildes rīku klasifikācija. Rīku klasifikācijai tika izvēlēti četri kritēriji: testa datu ieguves veids, skripta definēšanas veids, vienuma izmantošanas veids, vienuma interfeisa veids. Katrs kritērijs ietver vairākas klases, dažas no kurām iedalās apakšklasēs.

1.4.3. Izpētīti, analizēti un klasificēti pēc izvēlētajiem kritērijiem un klasēm 32 testu izpildes automatizācijas rīki. Šī rīku klasifikācija ļaus izvēlēties piemērotāko rīku konkrētas programmatūras testēšanai.

2. AUTOMATIZĒTO TESTU IZSTRĀDES PROCESI

2.1. Testu automatizācijas rīku projektēšana

Pirmajā darba nodaļā tika parādīts, ka pastāv daudzi dažādi testu automatizācijas rīki, piemēroti dažādu testēšanas uzdevumu veikšanai, un katrs ar savām niansēm. Neskatoties uz šo dažādību, joprojām bieži var būt situācijas, kurās esošie rīki nav piemēroti, un ir jāizstrādā jauns, uzdevumam specifisks, risinājums. Šajā apakšnodaļā tiek apkopoti vispārīgie apsvērumi, kurus ir nepieciešams ņemt vērā, izstrādājot jaunu rīku, un uz kuriem pamatojas darba ietvaros izstrādātie 4. un 5. nodaļās aprakstītie risinājumi.

2.1.1. Testu automatizācijas līmeņi

Testu automatizāciju var iedalīt vairākos līmeņos, piemēram, saskaņā ar [70]:

- Vienību līmenis (*unit*). Tiek atsevišķi testēta katra sistēmas vienība, pēc iespējas izolēti no citām, lai būtu vieglāk atrast tieši noteiktā vienībā esošas problēmas.
- Lietojumprogrammas saskarnes līmenis (*API*). Tiek testēta programmatūras atklātā API, lai nodrošinātu tās izmantošanas drošumu API klientu puses.
- Sistēmas līmenis (*system*). Tiek testēta sistēma kopumā, lai nodrošinātu to, ka visas saliktas kopā vienības funkcionē saskaņoti un ka sistēmas prasības ir apmierinātas.
- Lietotāja saskarnes līmenis (*user interface*). Sistēma vai tās modulis tiek testētas no gala lietotāja viedokļa, lai nodrošinātu to, ka programmatūra korekti reaģē uz lietotāja rīcībām.

Taču šis iedalījums ir skats no testēšanas mērķu viedokļa. No tehniskā viedokļa šie līmeņi var pārklāties. Piemēram, sistēmas līmeņa testēšanu var veikt gan ar lietotāja saskarnes, gan ar API palīdzību atkarībā no sistēmas dabas. API testēšana no tehniskā viedokļa bieži notiek tāpat kā testēšana vienību līmenī — funkciju izsaukšanas mehānisms var būt identisks. Tāpēc no testu automatizācijas rīku projektēšanas viedokļa būtu saprātīgāk apskatīt šādus līmeņus atbilstoši pirmajā nodaļā esošajai klasifikācijai pēc saskarnes veida (klases S1, S2 un S3).

- Lietotāja saskarnes līmenis (*UI*) (atbilst klasei S1). Tiek automatizētas darbības, kuras veic lietotājs, lietojot kādu vizuālu — grafisko jeb GUI (*Graphical User Interface*) vai konsoles jeb (*Console User Interface*) saskarni.
- Funkciju izsaukumu līmenis (atbilst klasei S2). Tiek automatizētas darbības, kuras veic citi moduļi (vai sistēmas) ar testējamo moduli (vai sistēmu), izpildot funkciju izsaukumus vai nu tieši, vai caur kādu augstāka līmeņa saskarni, piemēram, COM vai CORBA.
- Komunikācijas līmenis (atbilst klasei S3). Tiek automatizētas darbības, ko veic citi moduļi ar testējamo moduli, lietojot kādu komunikācijas protokolu, piemēram, tīklā (Ethernet, TCP, HTTP, utt.) vai tiešā savienojumā (COM, LPT, USB, utt.).

Tagad sīkāk izskatīsim šiem līmeņiem raksturīgus testu automatizācijas rīku projektēšanas aspektus.

2.1.2. Vienībtestēšanas rīku projektēšana

Vienībtestēšanas rīki strādā funkciju izsaukumu līmenī. Atkarībā no izvēlētiem testējamiem vienumiem un testu uzbūves tie var nodrošināt gan pašu vienībtestēšanu, gan integrācijtestēšanu, gan API testēšanu, gan (atsevišķos gadījumos) sistēmtestēšanu.

Vienībtestēšanas rīki ir visvienkāršākie no realizācijas viedokļa, jo nav jāizstrādā speciāls mehānisms sadarbībai ar testējamo vienumu. Praktiski katrai programmēšanas valodai ir pieejami savi rīki, kas prot veikt tiešos šajā valodā rakstīto funkciju izsaukumus. Pie tam šie rīki ir konceptuāli līdzīgi, jo ir atvasināti no Java vienībtestēšanas rīka JUnit, ko atspoguļo arī to nosaukumi, piemēram, CppUnit C++ valodai [28], PyUnit Python valodai vai NUnit .NET valodām [93]. Kopumā šo rīku saimi sauc par xUnit rīkiem [49].

Tātad vienībtestēšanas rīku īpašības ir vairāk saistītas ar pašu testu izstrādes un organizācijas atbalstu. Saskaņā ar [56] galvenās prasības pret vienībtestēšanas rīku ir sekojošas.

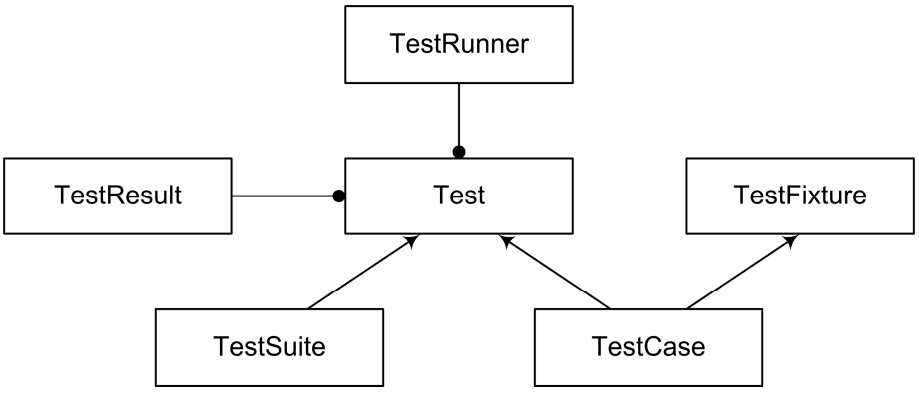
1. Testu kodam jābūt atdalītam no programmatūras koda. Kaut arī testi var tapt vienlaicīgi ar testējamo programmatūru, tiem nav jābūt pašas programmatūras sastāvdaļai.
2. Katram testam ir jābūt spējīgam izpildīties neatkarīgi no citiem. Tas nozīmē, ka testu komplektu sastāvs un testu secība tajos neietekmē katra atsevišķa testa izpildes rezultātus.

3. Testus ir jābūt iespējai patvaļīgi grupēt testu komplektos. Tātad testu jāvar pievienot jebkuram komplektam (vai pat vairākiem komplektiem) neatkarīgi no tā īpašībām.
4. Testa izpildes sekmīgumam vai nesekmīgumam ir jābūt uzreiz viegli saredzamam. T.i., rezultātiem jāparādās tādā veidā, lai nebūtu jāmeklē vai jāpēta, vai tests izgāja vai neizgāja.

Jāatzīmē, ka komunikācijas un it īpaši lietotāja saskarnes līmeņa testiem 2. un 3. īpašību nodrošināšana var būt neracionāla gan no pašu testu sarežģītības, gan no to izpildes ātruma viedokļa. Tādā gadījumā starp testiem var pastāvēt atkarības, kas var apgrūtināt testu komplektu izveidi. Šī darba 4. nodaļā tiek aplūkotas izstrādātās metodes un risinājumi, kas palīdz automatizēt šo procesu.

xUnit rīki kļuva par de facto standartu vienībtestēšanā, tāpēc jaunveidojamo vienībtestēšanas rīku arhitektūra visbiežāk atbilst JUnit arhitektūrai (gadījumos, kad ir jāizveido atbalsts kādai jaunai valodai), vai to paplašina, lai pievienotu tai papildus iespējas, piemēram, TestNG pievieno iespēju definēt testu savstarpējas atkarības, veikt parametrizētu testēšanu un arī citas iespējas testēt vairāku vienību sadarbības [17].

Ņemot vērā tādu xUnit lomu vienībtestēšanas rīku pasaulē, izskatīsim sīkāk šīs rīku saimes abstrakto arhitektūru pēc [49], kura ir atspoguļota 2.1. attēlā.



2.1. att. xUnit abstraktā arhitektūra

Attēla centrā ir Test elements, kas atbilst abstraktajam testam. Paši testi nav rīka elementi, bet tiek izstrādāti atbilstoši rīka nodrošinātajam Test interfeisam, abstraktajai klasei vai saskaņā ar kādu citu no valodas atkarīgu mehānismu.

Testi iedalās testu komplektos un testpiemēros, kur testpiemērs ir viens atsevišķs nedalāms tests (piemēram, atsevišķa TestCase interfeisu realizējoša klase), bet testu komplekts ir šādu testpiemēru virkne (ko izstrādā, piemēram, realizējot TestSuite interfeisu).

TestFixture ir abstraktā klase (vai interfeiss), kas nosaka testpiemēra klases struktūru. Parasti xUnit saimes rīkiem tā nosaka, ka TestCase realizējošajā klasē var būt metodes setUp() un tearDown(), kas izpildās attiecīgi pirms un pēc katra testpiemēra, lai nodrošinātu korektu testa vides inicializāciju un deinicializāciju.

TestRunner ir rīka modulis, kas atbild par testu izpildi. Tas ņem ieejā testu (kas var būt testpiemērs vai testu komplekts), izpilda to un fiksē rezultātus. TestResult modulis ir atbildīgs par rezultātu saņemšanu no TestRunner un tā atspoguļošanu lietotājam draudzīgā formā.

2.1.3. Komunikācijas testēšanas rīku projektēšana

Komunikācijas testēšana pievieno papildus sarežģītības dimensiju testēšanas rīku projektēšanai. Šī veida rīkiem ir jāatbalsta noteikts komunikācijas protokols, lai varētu darbināt testējamo vienumu.

Var iedalīt divus komunikācijas testēšanas rīku veidus.

1. Rīki, kas ir balstīti uz esošām vienībtestēšanas tehnoloģijām. Pēc būtības tie ir rīki, kas esošiem rīkiem, tādiem kā JUnit pievieno papildus iespējas tieši komunikācijas testēšanai. Tātad šādu rīku projektēšanas uzdevums reducējas uz protokolu atbalstoša moduļa izstrādi un tā integrāciju esošajā vienībtestēšanas risinājumā. Šāda rīka piemērs ir HttpUnit [37], kas nodrošina papildus funkcionalitāti komunikācijas testēšanai caur HTTP protokolu. Zemāka līmeņa protokolu (TCP vai UDP) komunikācijas testēšanai var izmantot arī pašu JUnit un Java standartbibliotēku.
2. Rīki, kas nav saistīti ar esošām vienībtestēšanas tehnoloģijām. Tā ir lielākā daļa komerciālo veiktspējas testēšanas rīku. Pie šādiem rīkiem ir pieskaitāmi TTCN-3 rīki [123]. TTCN-3 (*Testing and Test Control Notation*) ir standarts, kas apraksta speciālu tieši testēšanas mērķiem piemērotu programmēšanas valodu un arhitektūru, pēc kuras ir jāstrādā to atbalstošiem rīkiem [126].

Komunikācijas līmenī strādā vairākums veiktspējas testēšanas rīku [110]. Iemesls tam ir tas, ka daudz efektīvāk ir ģenerēt slodzi uz servera programmatūru, nevis palaižot vairākus klienta programmatūras eksemplārus (kas prasītu daudz datora resursu), bet emulējot komunikāciju no vairākiem vienlaicīgiem klientiem. Šādas emulēšanas nodrošināšanai arī ir nepieciešams savs modulis, kas spēj palaist un iteratīvi izpildīt vairākus testu eksemplārus vienlaicīgi.

Svarīgs aspekts komunikācijas līmeņa testēšanā ir arī komunikācijas pārtveršana. Tā var būt nepieciešama gan komunikācijas analīzei ar mērķi noskaidrot tās atbilstību protokola specifikācijai, kā arī kalpot par ievaddatiem veiktspējas testa skripta ģenerēšanai [100].

2.1.4. Lietotāja saskarnes testēšanas rīku projektēšana

Komandrindas saskarnes testēšana ir līdzīga komunikācijas testēšanai. Ir samērā viegli pārvirzīt lietojumprogrammas standarta ieejas un izejas plūsmas tā, lai testēšanas rīks varētu tās apstrādāt, kā parasto komunikācijas protokolu [81]. Līdz ar to, šeit tiks apskatīti rīki, kas nodarbojas ar grafiskās lietotāja saskarnes (GUI) testēšanu.

Lietotāja saskarnes testēšanas rīki ir ievērojami sarežģītāki par iepriekšējiem diviem rīku tipiem. Jaunas sarežģītības, ar ko sastopas šādu rīku izstrādātāji, ir vairākas (saskaņā ar [81]).

- Grafiskajai lietotāja saskarnei ir stāvoklis. Atkarībā no stāvokļa dažādu darbību veikšana var būt un var nebūt iespējama, piemēram, nevar nospiest pogu, kas ir deaktivēta. Atkarībā no stāvokļa viena un tā pati darbība var novest pie atšķirīga rezultāta.
- Rīkam ir jāprot veikt vairākas netriviāli automatizējamas darbības — spiest pogas, aizpildīt vai nolasīt ievadlaukus, pārvietot kursoru, veikt vilkšanas un nomešanas (*drag and drop*) operācijas utt.
- Rīkam ir jāprot atpazīt vizuālos objektus. Ja testā ir iekļauta darbība „nospiest pogu OK“, rīkam pirms darbības veikšanas šī poga ir jāatšķir ne tikai no ievadlaukiem, sarakstiem, rīkjoslām un citiem objektu veidiem, bet arī no citām līdzīgām pogām, kas atrodas uz ekrāna. Tehniski tas sarežģījas ar to, ka ir vairākas grafisko saskarņu tehnoloģijas un katrai ir nepieciešams savs atšķirīgs objektu atpazīšanas mehānisms.
- Rīkam ir jāprot reaģēt uz notikumiem, kas tiek atspoguļotas saskarnē. Piemēram, ja pogu var nospiest tikai kad parādās vajadzīgais logs un pati poga kļūst pieejama, rīkam jāprot šo momentu pamanīt.

Ievērojot šīs sarežģītības, esošie rīki ir ļoti dažādi gan pēc funkcionalitātes, gan pēc uzbūves [129]. Izvēloties šāda veida rīku (vai plānojot izstrādāt jaunu) ir jānovērtē, kādām īpašībām rīkam ir jāpiemīt. Iespējamo īpašību saraksts ir atrodams Dustin et al. grāmatā [32]. Dažas sarakstā atrodamas funkcionālās īpašības var tikt nodrošinātas ar atsevišķu rīka moduļu izstrādi. Un kaut arī lēmums par to, vai tie būs atsevišķi moduļi, vai funkcionalitāte tiks

nodrošināta savādāk, ir atkarīgs no vairākiem faktoriem un paliek projektētāja ziņā, uzskaitīsim šeit dažus kandidātmoduļus, kurus var apsvērt, projektējot lietotāja saskarnes testēšanas rīku.

- Ierakstīšanas modulis. Ierakstīšanas modulis ir nepieciešams plaši izplatītai ierakstīšanas-atspēlēšanas (*capture-replay*) funkcionalitātes nodrošināšanai. Tā izpaužas tā, ka aktivizēts modulis spēj „noklausīties", ko lietotājs dara ar testējamo lietojumprogrammu un automātiski ierakstīt tās darbības gatava skripta formā.
- Objektu īpašību apstrādes modulis. Nodrošina objektu atpazīšanas informācijas atsevišķu glabāšanu no skriptu teksta, kas palīdz ērtāk uzturēt testus, jo vizuālo objektu atpazīšanas loģiku var kontrolēt atsevišķi no testu procedūru loģikas.
- Vizuālo objektu tehnoloģiju atbalsta moduļi. Katrai vizuālo objektu tehnoloģijai (jeb videi) var tikt izstrādāts atsevišķs modulis, kas atbalsta šo objektu darbināšanu (un, iespējams, ierakstīšanu).
- Datubāzu verifikācijas modulis. Nodrošina iespēju veikt datubāzes stāvokļa verifikācijas kā testa soļus.
- Laika mērīšanas modulis. Ļauj nodrošināt atsevišķu (vai visu) darbību apstrādes ilgumu, kas ļauj novērtēt rīka darba ātrumu.
- Datu tabulu atbalsta modulis. Nodrošina iespēju atdalīt testa procedūru no testa datiem, uzturēt datus atsevišķi un izpildīt vairāku iterāciju testu ar dažādiem datu komplektiem.
- Rezultātu analīzes modulis. Ļauj fiksēt testa izpildes rezultātus, novērtē katra izpildīta soļa veiksmīgumu un neveiksmes gadījumu fiksē analīzei relevantu informāciju.
- Atskaišu ģenerēšanas modulis. Vairāku testu komplekta izpildes gadījumā nodrošina pārskatāmu atskaišu ģenerēšanu, kas palīdz ātri novērtēt testējamās sistēmas kvalitātes stāvokli.

Šo moduļu nepieciešamības novērtēšana, citu vajadzīgu moduļu identifikācija, kā arī to savstarpējo attiecību analīze ir viens no svarīgākajiem šādu rīku projektēšanas uzdevumiem.

2.2. Automatizētās testu programmatūras projektēšana

Kad rīks ir izvēlēts, vai arī izstrādāts no jauna, var sākties automatizēto testu izstrāde, ko rīks spēj darbināt. Ja testu ir daudz, tad izstrādājamos automatizēto testu komplektus mēdz saukt par testu programmatūru (*testware*), tādējādi pasvītrojot to, ka automatizētie testi ir programmatūras paveids ar līdzīgām prasībām pēc modularitātes, elastības, uzturamības, utt.

No otras puses, testu programmatūra tomēr ir specifisks programmatūras paveids, līdz ar to tai piemīt specifiskas īpašības.

Šajā sadaļā tiks izskatīti tipiskie testu programmatūras struktūras elementi, testu programmatūras analīzes un projektēšanas process, ka arī dažas testu automatizācijas heiristikas.

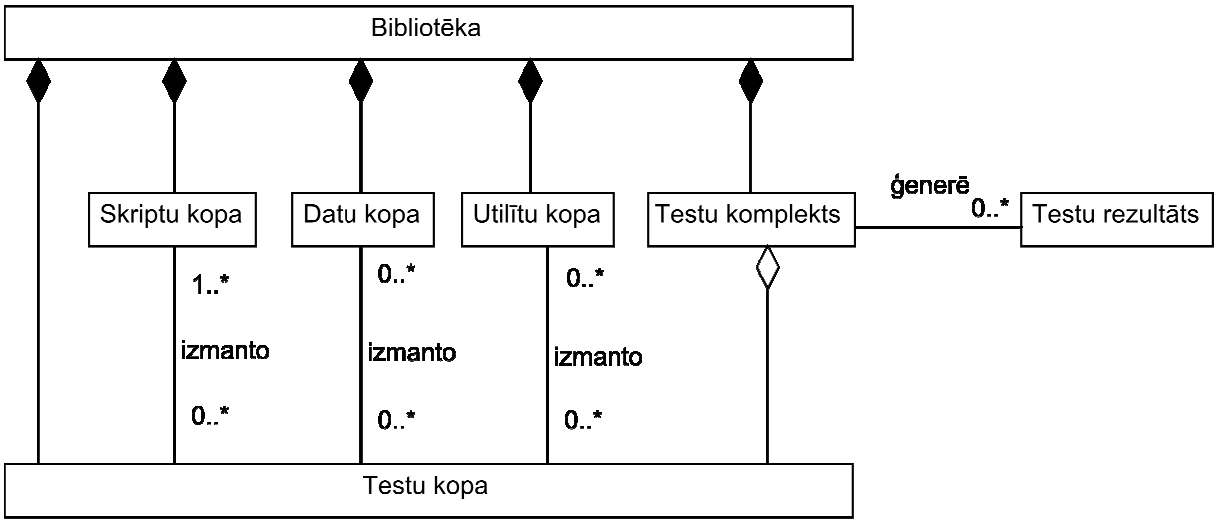
2.2.1. Testu programmatūras struktūra

Testu programmatūra saskaņā ar [39] sastāv no sekojošiem elementiem.

- Testu kopas (*test sets*). Testu kopa ir viens vai vairāki testpiemēri, kuri paredzēti kādas vienas testējamās programmatūras īpašības pārbaudei. Testu kopa integrē skriptu kopas, datu kopas, utilītu kopas, lai izveidotu darbināšanai gatavu testu.
- Skriptu kopas (*script sets*). Skripts nodrošina testa procedūras izpildi, t.i., definē, no kādiem soļiem sastāv tests. Skripts var tikt rakstīts kādā programmēšanas valodā vai tikt uzdots kādā citā veidā, piemēram, deklaratīvajā valodā vai vizuāli. Skriptu kopā viens vai vairāki cieši saistītie skripti, piemēram, atsevišķi papildus skripti var realizēt soļu virknes, kuras tiek izsauktas no galvenā skripta vairākām vietām.
- Datu kopas (*data sets*). Testa dati ir konkrētas vērtības, ko izmantos skripti, lai izpildītu testpiemērus. Datu kopa var saturēt datus vienam vai vairākiem testpiemēriem. Pateicoties skriptu un datu atdalīšanai, ar viena skripta palīdzību var viegli sagatavot vairākus testpiemērus (variējot datus), vai arī vienus un tos pašus datus var izmantot vairākos skriptos, tādējādi ievērojami palielinot testu pārklāšanas iespējas.
- Utilītu kopas (*utility sets*). Utilītas ir atkārtojami izmantojama funkcionalitāte, ko var izsaukt testu skripti. Tās iekļauj aizbāžņus, dzinējus, konvertorus, salīdzinātājus utt. Utilītu kopas ir izdevīgi atdalīt no skriptiem, jo tas atvieglo to atkārtoto izmantošanu un uzturēšanu.
- Testu komplekti (*test suites*). Testu komplekts ietver vienas vai vairāku testu kopu izsaukumus. Testu komplekts ir vienas testu izpildes sesijas objekts. Tātad testu komplektus veido vadoties pēc testēšanas mērķa, piemēram, apvienojot vissvarīgākos testus ātrai kvalitātes novērtēšanai (t.s. dūmu testus) vai testus, kas ir paredzēti viena konkrēta sistēmas aspekta testēšanai (piemēram, lietotāja kontu apstrādi). Visu izstrādātu testu izpildei arī var tikt izveidots atsevišķs testu komplekts, kas sastāvēs no visām testu kopām.

- Testu programmatūras bibliotēkas (*testware libraries*). Bibliotēka kalpo par repozitoriju, kas glabā visus pārējus vienumus. Piemēram, katrai testējamai sistēmai var tikt uzturēta sava testu programmatūras bibliotēka, kurā tiek glabātas visas tās sistēmas testēšanai izstrādātas skriptu, datu un testu kopas, kā arī testu komplekti.
- Testu rezultāti (*test results*). Testu komplekta izpildes laikā rodas testu rezultāti. Šo rezultātu ilglaicīga uzturēšana ar iespēju salīdzināt jaunākos rezultātus ar vecākajiem ir svarīga no kvalitātes kontroles viedokļa, jo atvieglo sekošanu līdzi testēšanas un defektu labošanas procesam.

Kopsavilkumam šo elementu attiecības attēlosim konceptuālajā UML klašu diagrammā 2.2. attēlā.



2.2. att. Testu programmatūras arhitektūra

Šo elementu fiziska izvietošana ir atkarīga no pielietota testu automatizācijas rīka, kā arī no testu projektētāja izvēles. Piemēram, tie var glabāties failu sistēmā, relāciju datubāzē, to izmaiņu izsekošanai var lietot versiju kontroles rīkus utt. Testu efektivitāte lielā mērā būs atkarīga no tā, kādi dati ir atdalīti no skriptiem, bet kādi paliek konstantas skripta sastāvdaļas, kāda funkcionalitāte ir iznesta utilītās, bet kāda paliek pašos skriptos. Testu kopu kombinēšana testu komplektos ir atkarīga no testēšanas mērķiem — kādi sistēmas aspekti būs jātestē un cik ātrai ir jābūt testu izpildei.

Atbilstoši [32] automatizēto testu programmatūras projektēšanas procesu var iedalīt divās fāzēs:

1) testēšanas prasību analīze, kuras rezultātā tiek izstrādāta testēšanas prasību matrica un novērtētas iespējamas testēšanas tehnikas;

2) testu programmatūras projektēšana, kuras rezultātā rodas testu procedūru definīcijas (specifikācijas) pietiekamā detalizācijas līmenī.

2.2.2. Testēšanas prasību analīzes process

Testēšanas analīzes process, saskaņā ar [32], sastāv no sekojošajiem soļiem:

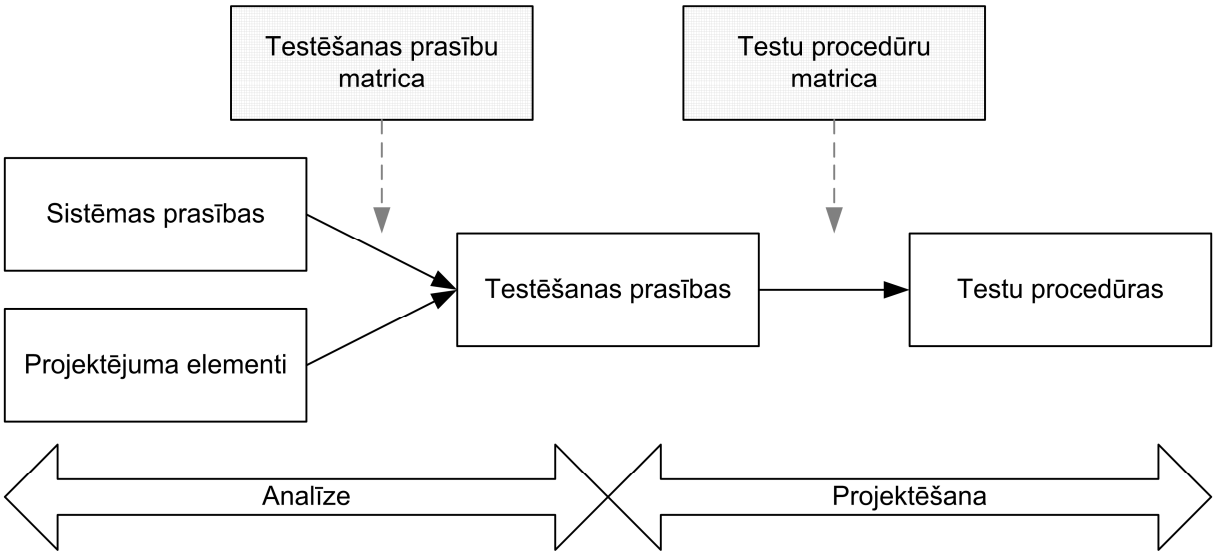
1. Mērķu analīze. Testētājiem jāizskata testēšanas mērķi un stratēģijas. Testēšanas mērķi un stratēģijas kopā ar prasībām pret testējamo sistēmu kalpo par primāro informāciju testēšanas projektēšanas procesam.
2. Verifikāciju metožu izvēle. Sistēmas prasībām vai lietošanas scenārijiem ir jāizvēlas verifikācijas metodes un jādokumentē izvēle prasību trasējamības matricā. Piemēram, prasības izpildi var novērtēt no lietotāja saskarnes viedokļa, no datubāzes lauku aizpildīšanas viedokļa, no ātrdarbības viedokļa utt. Šis solis zināmā mērā nosaka testu automatizācijas rīka izvēli (un arī ir ierobežots ar pieejamo rīku klāstu).
3. Testēšanas prasību analīze. Jāformulē testēšanas prasības, kas ir jāatvasina no testējamās sistēmas prasībām. Ja sistēmas prasības nosaka to, kādai ir jābūt sistēmai, tad testēšanas prasības nosaka, kādiem ir jābūt testiem, kas novērtē, vai sistēma nodrošina savu prasību izpildi.
4. Testēšanas prasību matricas izveide. Testēšanas prasības ir jākartē uz sistēmas prasībām un/vai sistēmas projektējuma komponentiem. Šāda matrica ļauj kontrolēt prasību (vai komponenšu) pārklājumu ar testu prasībām, t.i., nodrošināt, lai no testēšanas prasību izpildes konstatēšanas izrietu arī sistēmas prasību izpildīšanās pie iespējami minimāla testēšanas prasību skaita.
5. Testēšanas tehniku kartēšana. Jāveic sākotnēja testēšanas prasību saistīšana ar testēšanas tehnikām. Testēšanas prasību matrica var tikt mainīta, lai panāktu atbilstību ar šo kartēšanu. Testēšanas prasības lielā mērā nosaka testēšanas tehnikas, piemēram, prasība „testam jākonstatē, vai sistēmas reakcijas laiks paliek mazāks par 2 sekundēm, kad ar to vienlaicīgi strādā 50 lietotāji“ acīmredzot pieprasa veiktspējas testēšanas tehnikas lietošanu.

2.2.3. Testu programmatūras projektēšanas process

Testu programmatūras projektēšanas process saskaņā ar [32] sastāv no sekojošajiem soļiem.

1. Testu programmatūras modeļa definēšana. Jānovērtē dažādu testēšanas tehniku piemērotība testējamai programmatūrai un to atbilstība testēšanas prasībām. Tiek analizētas no testēšanas prasībām atvasinātas testēšanas tehnikas, kā un cik labi tās papildina viena otru, un vai nevar šo tehniku izmantošanu izvērst arī uz citu testēšanas prasību labāku nodrošināšanu. Rezultātā jātop skaidram, kādas konkrētas tehnikas tiks izmantotas un cik lielā mērā, ar šo informāciju papildinot testēšanas prasību matricu.
2. Testu arhitektūras definēšana. Jāizvēlas testu arhitektūras modelis un jānodefinē šī modeļa atribūti. Šo arhitektūru var būvēt balstoties uz testējamās sistēmas projektējumu (testi tiks grupēti pēc tā, uz kāda sistēmas elementa testēšanu tie galvenokārt ir orientēti) vai uz pielietojamām tehnikām (testi tiks grupēti pēc tā, kādu testēšanas tehniku tie realizē), vai arī kombinējot abas pieejas. Jāpieņem lēmums par to, kādās attiecībās būs automatizētie testi un manuālie — vai tie glabāsies kopā, vai atsevišķi, vai un kādā veidā būs iespējams pārtaisīt manuālo testu par automatizēto.
3. Testu procedūru definēšana. Jādefinē testu procedūru loģiskās grupas atbilstoši izvēlētajai testu arhitektūrai. Tiek izvēlēts testa procedūras formāts — kādiem standartiem ir jāatbilst procedūras aprakstam, cik detalizēti tie būs jāapraksta, kādi papildus atribūti ir jādefinē procedūrām.
4. Testu procedūru kartēšana. Jānovērtē attiecības starp testu procedūras un testēšanas prasībām. Tiek izveidota atbilstoša testu procedūru matrica. Matrica ļauj kontrolēt testēšanas prasību pārklājumu ar testu procedūrām, lai varētu nodrošināt pietiekamu pārklājumu no vienas puses un minimālu testu procedūru skaitu no otras puses.
5. Automatizēto vai manuālo testu kartēšana. Jāizvēlas, kādas testu procedūras ir jāautomatizē. Testu procedūru matrica tiek papildināta ar šo informāciju, novērtējot arī to, kādas testu procedūru daļas var tikt atkārtoti izmantotas citās procedūrās (utilītu definēšana).
6. Testu datu kartēšana. Katrai procedūrai jānovērtē, kādi datu veidi tai ir nepieciešami. Testu procedūru matrica tiek papildināta ar šo informāciju.

Kopsavilkumam testēšanas prasību analīzes un testu programmatūras projektēšanas procesi tiek ilustrēti 2.3. attēlā.



2.3. att. Testu prasību analīze un testu programmatūras projektēšana

Aiz testu programmatūras projektēšanas loģiski seko testu konstrukcijas fāze, kad definētajām testu procedūrām tiek izstrādāti skripti, tiek noformētas testu kopas un testu komplekti, kuri ir gatavi darbināšanai.

2.2.4. Testu automatizācijas heiristikas

Kā jebkurā projektēšanas sfērā, testu programmatūras projektēšanā ir savas heiristikas, kas veicina ideju ģenerēšanu optimālāku pieeju izvēlei. Tikai grāmatā [71] ir uzskaitīti 40 šādi padomi (apsvērumi), noderīgākie no kuriem pēc darba autora pieredzes ir sekojoši.

- Simtprocentīga automatizācija nav mērķis. Ne visus testus var automatizēt, piemēram, lietojamības testi, kas pārbauda programmatūru no tās lietošanas ērtuma viedokļa, nav automatizējami. Pie tam automatizācija prasa ievērojamu laika ieguldījumu un atsevišķus testus var būt izdevīgāk izpildīt manuāli. Jāautomatizē tikai tie testi, kas reāli atmaksājas.
- Testēšanas rīks nav stratēģija. Rīki nenosaka kā ir jātestē. Testēšanas stratēģijas izvēle ir augstāka līmeņa lēmums nekā automatizācijas rīka izvēle. Rīks ir tikai rīks un tas neaizvieto testētājus, testu projektētājus un testēšanas vadītājus.
- Nav jāautomatizē haoss. Sliktu testu automatizācija ļaus tikai paātrināt sliktu testēšanu. Automatizēto testu projektēšana ir pakļauta testu projektēšanai kā tādai, nevis aizvieto to. Kamēr testēšanas process nav pienācīgi organizēts, testu automatizācijai nav lielas vērtības.

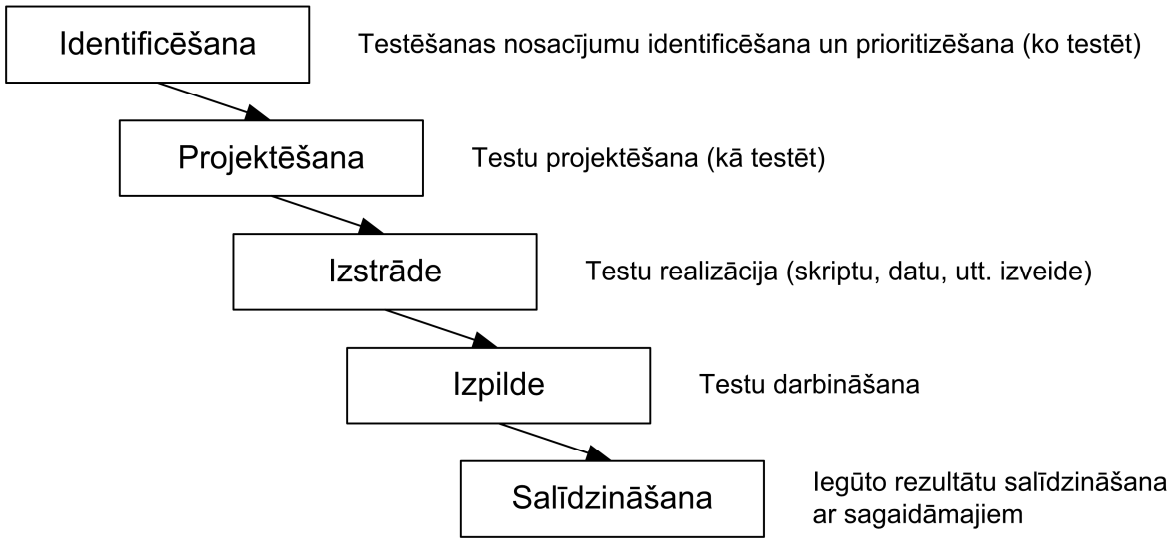
- Tīra ierakstīšana/atspēlēšana nestrādā. Ierakstīšanas/atspēlēšanas iespēju demonstrācija var radīt iespaidu, ka testu automatizācija ir viegls process. Taču vienkārši ierakstīts skripts ir ļoti jūtīgs pret minimālām saskarnes izmaiņām, tāpēc ļoti ātri noveco. Projektējot automatizētos testus, šis aspekts ir jāņem vērā un jāminimizē šis jūtīgums.
- Saskarnes mainās. Programmatūrai attīstoties mainās gan lietotāja, gan iekšējas saskarnes. Skripti, kas darbina programmatūru caur tām saskarnēm, šajos brīžos beidz pareizi strādāt. Šis aspekts jāņem vērā, lai pēc iespējas atdalītu to testu kodu (vai datus), kas ir jūtīgs pret šādām izmaiņām, no koda kas nav no tām saskarnēm atkarīgs, lai atvieglotu skriptu uzturēšanu.
- Automatizētie testi noveco. Neatkarīgi no tā, cik labi pret izmaiņām jūtīgs kods ir atdalīts no stabila, daži automatizētie testi novecos, piemēram, mainoties sistēmas prasībām. Pirms lēmuma pieņemšanas par testa procedūras automatizāciju ir jānovērtē skripta iespējamais dzīves ilgums, lai pārliecinātos par tā izstrādes izdevīgumu.
- Jāizvairās no sarežģītas skriptu loģikas. Loģika (nosacījumu izteiksmes) skriptā padara to par grūtāk saprotamu un mazāk drošu. Noteikti ir jāizvairās no testējamas programmatūras loģikas atkārtošanas testu programmatūrā. Retos gadījumos, kad loģika ir nepieciešama (vides konfigurācijas nodrošināšana, sarežģītu notikumu apstrāde), šo kodu labāk ievietot atsevišķajās funkcijās, pamata testus saglabājot tik lineārus, cik iespējams.
- Nav lieki jāizvairās no koda atkārtošanas. Parastajā programmēšanā valda princips, ka atkārtojams kods ir jāievieto atsevišķās funkcijās (procedūrās, apakšprogrammās). Testu skriptos šis princips nav vietā, jo testa soļu virknes ir cieši piesaistītas konkrētam skriptam, un soļu secību izmaiņām vienā skriptā nav jāietekmē citi skripti. Tātad tikai koda atkārtošanās nav iemesls to izolācijai atsevišķajās funkcijās.
- Ar datiem vadāmo testu automatizācija ir efektīva. Ja ir iespējams vienam skriptam piesaistīt vairākus datu komplektus, tas ievērojami palielina testu automatizācijas efektivitāti, jo jauna datu komplekta izveide ir vienkāršāka par jauna skripta izveidi.
- Jāapsver testa datu ģenerēšanas iespējas. Dažos gadījumos var būt izdevīgi nevis manuāli definēt testa datus, bet ģenerēt tos automātiski. Starp šiem gadījumiem var minēt nepieciešamību pēc lieliem ievadfailiem, pilnu ievaddatu kombināciju pārlasi, pāru vai ekvivalenču klašu testēšanu un citus.
- Jādod priekšroka iekšējo saskarņu testiem. Ja kādas funkcionalitātes testus var automatizēt gan lietotāja saskarnes, gan iekšējas saskarnes līmenī, priekšroka jādod

iekšējām saskarnēm, jo šādus skriptus ir vieglāk izstrādāt un uzturēt, un parasti tie arī izpildās ātrāk.

- Testējamības nodrošināšana atmaksājas. Ir vieglāk nodrošināt testējamību pašā testējamajā sistēmā, atvieglojot turpmāko testu automatizāciju, nekā automatizēt sistēmu, kas grūti pakļaujas automatizācijai.
- Jāsāk automatizēt agrāk. Agrīnā automatizētās testēšanas uzsākšana ļauj ātrāk identificēt testējamības problēmas un ieviest attiecīgas izmaiņas sistēmas projektējumā. Daži automatizētās testēšanas darbi, ieskaitot testu programmatūras projektēšanu un no saskarnēm neatkarīga skripta koda izstrādi, var tikt uzsāktas vēl pirms parādās pirmās darbināmas sistēmas versijas.

2.3. Testu automatizācijas procesu modeļi

Pastāv vairāki automatizētās testēšanas procesa modeļi. M. Fewster un D. Graham piedāvā procesa modeli, kas sastāv no piecām fāzēm [39]. Šis modelis ir shematiski atspoguļots 2.4. attēlā.



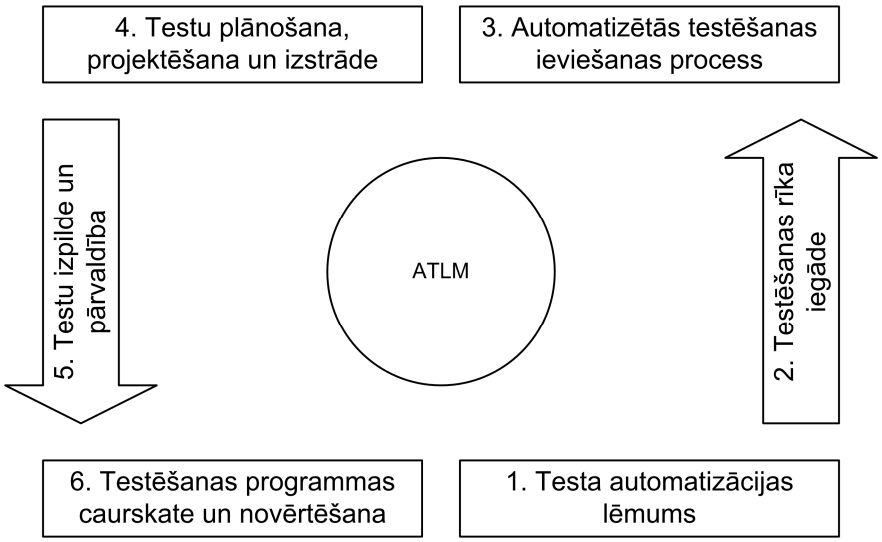
2.4. att. Automatizētās testēšanas process

Šī procesa fāzes ir sekojošas.

1. Testēšanas nosacījumu identificēšana. Tā ir sākotnēja sistēmas prasību analīze ar mērķi izveidot testēšanas prasības. Testēšanas prasības ir šīs stadijas rezultāts, taču tās var tikt precizētas projektēšanas stadijā.

2. Testu projektēšana. Šajā stadijā tiek izvēlēta testu arhitektūra, izmantojamās testēšanas tehnikas un rīki, tiek identificētas testu procedūras un testpiemēri, kas ir jāautomatizē.
3. Testu izstrāde. Pēc izstrādātā projektējuma testi ir jārealizē, izmantojot noteiktus rīkus. Šīs stadijas rezultāts ir darbināšanai gatavi skripti ar konkrētiem izmantošanai gataviem testa datiem, kas ir apvienoti darbināmos testu komplektos.
4. Testu izpilde. Testu komplekti tiek automātiski izpildīti pēc testētāja komandas. Rezultātā rodas testu izpildes rezultāti.
5. Rezultātu salīdzināšana. Testā iegūtie rezultāti tiek salīdzināti ar sagaidāmajiem un tiek novērtēts testa izpildes veiksmīgums vai neveiksmīgums. Parasti rīki šo uzdevumu arī izpilda automātiski vai nu testa izpildes laikā, vai uzreiz pēc testu izpildes.

E. Dustin et al. piedāvā automatizētā testa dzīvescikla metodoloģiju (*ATLM, Automated Test Life-Cycle Methodology*) [32], kas sastāv no sešām fāzēm, kuras ir atspoguļotas 2.5. attēlā.



2.5. att. ATLM procesa stadijas

ATLM fāzes ir sekojošas:

1. Testa automatizācijas lēmums. Šajā stadijā ir jāapzinās testēšanas mērķi un jānovērtē automatizācijas piemērotība un priekšrocības. Ir arī jāapzinās vai ir pieejami rīki dotā testēšanas uzdevuma veikšanai un kādi tie ir.
2. Testēšanas rīka iegāde. Kad ir pieņemts lēmums testu automatizēt, ir jāizvēlas piemērotākais rīks: ir jāizskata esošas alternatīvas un jāizmēģina labākie varianti. Tad rīks, kas izrādās vislabākais dotā uzdevuma veikšanai, ir jāiegādājas un jāsagatavo darbam.

3. Automatizētās testēšanas ieviešanas process. Kad rīks ir izvēlēts, tā izmantošana ir jāintegrē testēšanas procesā. Ja ir nepieciešams, ir adaptē testēšanas process rīka izmantošanai, jāmaina testētāju lomas, jāveic apmācība, jāpielāgo projekta laika grafiks.
4. Testu plānošana, projektēšana un izstrāde. Šajā stadijā notiek detalizētā testu plānošana, automatizēto testu projektēšana un izstrāde.
5. Testu izpilde un pārvaldība. Kad testi ir izstrādāti un testējamā programma ir piegādāta testēšanai, testi ir jāizpilda. Testu izpildes rezultātu analīze parasti noved pie defektu atklāšanas, līdz ar to šī fāze var sastāvēt no vairākām atkārtotām izpildēm.
6. Testēšanas un programmas caurskate un novērtēšana. Šajā stadijā, tiek apkopoti visu iepriekšējo stadiju rezultāti, novērtēta reālā atdeve no automatizācijas, un uz secinājumu pamata testēšanas automatizācijas stratēģija tiek adaptēta nākamajiem projektiem.

Salīdzinot šos divus modeļus, var secināt, ka Fewster-Graham procesa modelis ir šaurāks par ATLM un pārklāj tikai 4. un 5. ATLM fāzes.

2.4. Vienots automatizētās testēšanas modelis

Abos izskatītajos modeļos ir tāds trūkums, ka tie ignorē testu automātiskās ģenerēšanas iespēju, pieņemot ka visi automatizētie testi tiks izstrādāti manuāli. Lai novērstu šo trūkumu tika izstrādāts vienots automatizētās testēšanas modelis, kas iekļauj arī testu ģenerēšanas iespēju.

Modeļa elementi ir aktivitātes un komponenti. Komponenti ir darba produkti, kas tiek radīti testēšanas procesa gaitā. Modeļa aktivitātes ir darbības, kas ir veicamās manuāli vai automatizēti, kuru rezultātā no viena veida komponentiem tiek iegūti citi.

Vienotais automatizētās testēšanas modelis ietver sekojošas aktivitātes:

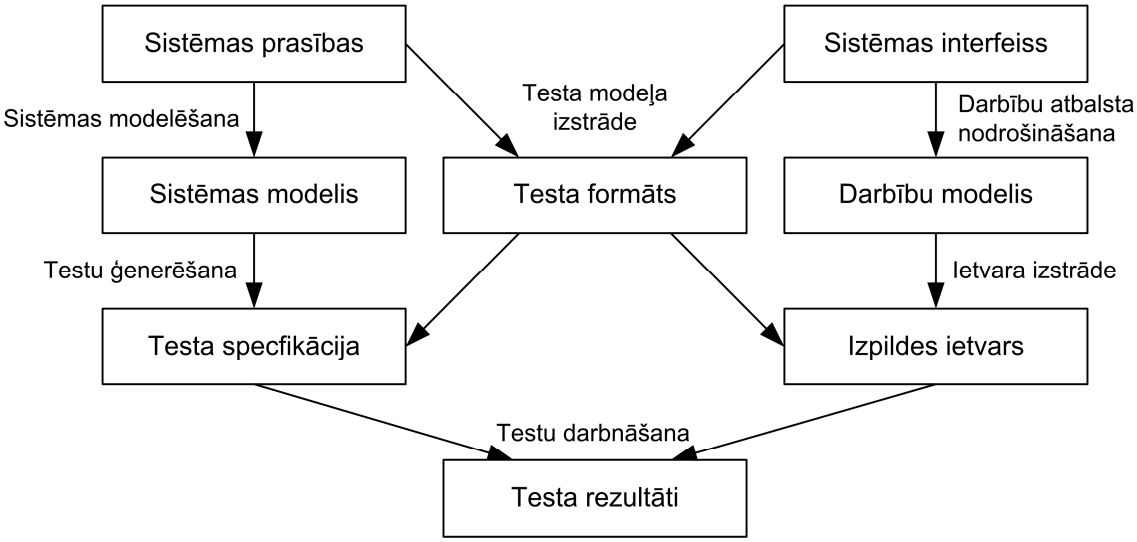
- testa modeļa izstrāde;
- sistēmas modelēšana;
- darbību atbalsta nodrošināšana;
- testu ģenerēšana;
- ietvara izstrāde;
- testu darbināšana.

Minētās aktivitātes darbojas ar sekojošiem modeļa komponentiem:

- sistēmas prasības;
- sistēmas interfeiss;

- testa formāts;
- sistēmas modelis;
- darbību modelis;
- testa specifikācija;
- izpildes ietvars;
- testa rezultāti.

Saistības starp modeļa aktivitātēm un komponentiem ir parādītas 2.6. attēlā.



2.6. att. Vienots automatizētās testēšanas modelis

Šīs apakšnodaļas turpinājumā tiek aprakstīts modeļa struktūras pamatojums un tā lietošanas iespējas.

2.4.1. Automatizētās testēšanas modeļa komponenti un aktivitātes

Sistēmas prasības ir visi informācijas avoti, no kuriem seko testēšanas prasības, un uz kuru pamata var izstrādāt testpiemērus. Testēšanā ir jānovērtē sistēmas atbilstība prasībām, lai pārliecinātos par to, ka tā pasūtītāja vēlmes ir apmierinātas.

Sistēmas interfeiss ir vai nu gatavs sistēmas interfeiss, vai tikai projektējums, vai vispārīgi — precīza informācija par to, kādi būs interfeisa elementi un kādā veidā tie būs lietojami. Šī informācija var izrietēt no sistēmas prasībām, bet var arī tieši neizrietēt, ja sistēma ir kādas lielākas sistēmas modulis.

Minētie komponenti ir sistēmas pamats, testēšanas procesa ievades, jo automatizētajiem testiem no vienas puses ir jānovērtē sistēmas prasību izpildi, bet no otrās puses — tas jādara darbojoties ar sistēmas interfeisu (lietotāja saskarnes, komunikācijas vai funkciju izsaukumu līmenī).

Testa rezultāti ir tas, kas ir jāiegūst beigās, testēšanas procesa izvade.

Tātad vienotais automatizētās testēšanas modelis apraksta procesu, kādā no sistēmas prasībām un sistēmas interfeisa tiek iegūti testa rezultāti, kas kalpo par pamatu sistēmas kvalitātes novērtējumam. Modeļa aktivitātes ir darbības, kas kopumā virza procesu uz šo mērķi, bet pārējie komponenti ir šī procesa starpprodukti.

Testa modeļa izstrāde ir aktivitāte, kuras rezultātā uz sistēmas prasību un sistēmas interfeisa pamata tiek izstrādāts (vai izvēlēts) testa modelis, kas nosaka testa struktūru. Tiek izvēlēts arī formāts, kādā gatavam testam ir jābūt pierakstītam. Šī aktivitāte nav automatizējama, taču prasa salīdzinoši maz laika resursu.

Testa formāts ir komponents, kas rodas testa modeļa izstrādes aktivitātes rezultātā. Šis formāts ir sava veida kontrakts starp testa izstrādātāju vai ģeneratoru no vienas puses un testa izpildītāju vai automātiskās izpildes rīku vai ietvaru no otrās puses. Testu, kas atbilst šim formātam, būs iespējams automātiski darbināt.

Sistēmas modelēšana ir aktivitāte, kas no sistēmas prasībām izveido sistēmas modeli. Sistēmas prasības ir jāatspoguļo tādā formā, no kuras būtu viegli atvasināt testpiemērus. Tipiski šis process nav automatizējams, taču ja sākotnēji sistēmas prasības ir veidotas kādā formālajā notācijā, ko var interpretēt programmatūra, to var ņemt par sistēmas modeli bez izmaiņām vai arī transformēt, pilnībā vai daļēji automatizējot šo procesu.

Sistēmas modelis ir kādā formālajā notācija veidots skats uz sistēmu. Modeļa formalitātes pakāpē ir atkarīga no tā, vai testpiemēru ģenerēšana notiks automātiski vai nē. Automātiskās ģenerēšanas gadījumā modelim ir jābūt ar datorprogrammām apstrādājamam, un tas var kalpot par ievadu formālām testpiemēru projektēšanas metodēm, vairākas no kurām tika aplūkotas 1.2. sadaļā. Par modeļa veidu ir uzskatāms arī programmas pirmkods, tādā gadījumā testpiemērus var ģenerēt ar baltās kastes ģenerēšanas metodēm.

Testu ģenerēšana ir testu specifikāciju izveides aktivitāte, kas var būt automātiska, ja sistēmas modeļi to pieļauj, vai manuāla. Automātiskās testu ģenerēšanas gadījumā tiek izmantota speciālā programmatūra — testu ģenerators. Var izmantot kādu no vairākiem esošiem ģeneratoriem vai izstrādāt jaunu (tādā gadījumā tas ir atsevišķs process un šajā modelī nav iekļauts). Ģenerējot vai izstrādājot testa specifikāciju, jāvadās no sistēmas modeļa no vienas puses un no izvēlētā testa formāta no otrās puses.

Testa specifikācija ir komponents, kas ir testa formātam atbilstošs testa apraksts. Tas ir testu ģenerēšanas aktivitātes rezultāts.

Darbību atbalsta nodrošināšana ir aktivitāte, kurā testēšanas veikšanai nepieciešamas darbības ar sistēmas interfeisu tiek automatizētas, t.i., tiek nodrošināta šo darbību veikšana ar programmatūras līdzekļiem. Darbību kopums ir atkarīgs no sistēmas interfeisa (ko vispār var veikt ar sistēmu) un testu formāta (kādas darbības būs jāveic izpildot šāda veida testus). Rezultātā tiek veidots darbību modelis. Šī aktivitāte nav automatizējama, jo ietver programmēšanas darbu, bet tās darbietilpība ir atkarīga no testējamā sistēmas interfeisa, piemēram, API interfeisam tā ir triviāla, bet lietotāja saskarnes gadījumā tā var prasīt ievērojamu laiku.

Darbību modelis ir komponents kas rodas darbību atbalsta nodrošināšanas aktivitātes rezultātā. Darbību modelis var būt skriptu vai funkciju bibliotēkas formā, tas nodrošina atomāro darbību veikšanu pret sistēmas interfeisu. API līmeņa sistēmas interfeisa gadījumā, pati API var kalpot par darbību modeli. Vizuālās saskarnes gadījumā tā ir jāanalizē lai identificētu iespējamas darbības un jāimplementē attiecīgas darbināšanas funkcijas.

Ietvara izstrāde ir aktivitāte kurā tiek izstrādāts ietvars jeb testu dzinis jeb testu izpildes rīks, kas spēj automātiski darbināt testus, atbilstošus izvēlētajam testa formātam, veicot darbību modelī nodrošinātas darbības. Dažādos gadījumos ir iespējams izmantot kādu esošu rīku, vairāki no kuriem ir minēti 1.3. sadaļā, iespējams, pielāgojot to, vai arī izstrādāt jaunu rīku, vadoties pēc 2.1. sadaļā aplūkotajām vadlīnijām.

Izpildes ietvars ir programmatūra, kas nodrošina testu specifikāciju automātisku izpildi, ja vien testa specifikācija atbilst testa formātam.

Testu darbināšana ir izveidoto testa specifikāciju automātiskās izpildes aktivitāte. Izpildes ietvaram interpretējot testa specifikācijas, veic darbības ar sistēmu, izmantojot darbības modelī nodrošinātās iespējas.

Testa rezultāti ir rezultāti, kas tiek iegūti testu izpildes gaitā.

2.4.2. Automatizētās testēšanas modeļa pielietošanas iespējas

Modeļa realizācija ir atkarīga no vairākiem aspektiem, tādiem kā testēšanas mērķi, testējamās sistēmas īpatnības un citiem. Modelis ir elastīgs tādā ziņā, ka ļauj atsevišķām aktivitātēm būt gan manuālām, gan automatizētām, ar izņēmumu, ka testu darbināšanai ir

jābūt automātiskai, jo tas ir paredzēts automatizētās testēšanas modelēšanai. No otrās puses, darbību atbalsta nodrošināšana un jo vairāk ietvara izstrāde ir neautomatizējamas aktivitātes.

Modeļa komponentu un aktivitāšu realizācija ir atkarīga arī no tā, kāda veida rīks (pēc 1.3. sadaļā aplūkotās klasifikācijas) tiek izmantots kā izpildes ietvars.

D1 klases rīkiem, kuriem testa dati ir neatņemama skripta sastāvdaļa, par testa specifikāciju var kalpot pats skripts. Testu ģenerēšana varētu tikt nodrošināta, balstoties uz skripta šabloniem, ja rīks ir S1 vai S2 klasē, t.i., skripts ir definējams programmēšanas vai deklaratīvajā valodā. Ģenerators no šablona var izveidot vairākus skriptus, kas atbilst dažādiem testpiemēriem. Rīks (izpildes ietvars) šos skriptus paņem kā testu specifikācijas un interpretē, iegūstot testa rezultātus.

D2 un D3 klases rīkiem (dati ir atdalāmi no skripta) par testa specifikācijām var kalpot arī testa dati — ievaddati un sagaidāmie rezultāti. Izpildāmais skripts, varētu būt uzskatāms par izpildes ietvara sastāvdaļu.

S1 klases rīkiem, kuriem skripts ir veidojams programmēšanas valodā, vispār ir plašākas iespējas un vairākas alternatīvas, jo attiecīgajos skriptos ir iespējams realizēt sarežģītāku testa specifikācijas apstrādes loģiku. Ar šādu rīku palīdzību ir iespējams izstrādāt jaudīgus testu dziņus, kuri prot interpretēt sarežģīta formāta testu specifikācijas, kas ir īpaši piemērotas sistēmas darbību atspoguļošanai. Tādā veidā ir iespējams nodrošināt ar atslēgas vārdiem vadāmo testēšanu (*keyword-driven testing*), piemērotu specifiskajām testējamo sistēmu vajadzībām [112, 117].

S2 klases rīkiem (skripts veidojams deklaratīvajā valodā) tik plašu iespēju nav, testu specifikāciju formāts tiem ir ierobežots ar testa datiem vai testa skriptiem. To gan var paplašināt izmantojot papildus programmatūru, piemēram, nodrošinot testa specifikācijas transformēšanu rīkam saprotamajā formā, tādējādi šo uzdevumu pārnesot no testu ģeneratora uz izpildes ietvaru.

Īpašs gadījums rīki kas pieder klasei S3 un D1 (vizuāli veidojami skripti, kuriem testa dati ir neatņemama sastāvdaļa). Šāda veida rīkiem nav iespējams nodrošināt automātisku testu ģenerēšanu, visi testi jāveido manuāli.

Rīku piederība klasēm pēc M un I kritērijiem (vienuma izmantošanas veids un vienuma interfeisa veids) ietekmē tikai izpildes ietvara un darbību modeļa realizācijas detaļas, tās mazāk iespaido testa specifikācijas formātu, izņemot gadījumus, kad tas ir izdevīgi testējamās sistēmas īpatnību dēļ.

Modelis ir izmantojams ne tikai testu ģenerēšanas un izpildes uzdevumā, bet arī testu komplektu ģenerēšanā un izpildē. Par testu specifikāciju tādā gadījumā kalpo testu secība

komplektā, testu ģenerators izveido šo secību, vadoties pēc esošu testu īpašībām, bet testu ietvara loma ir testu dzinim, kas šo secību izpilda. Detalizētāk šī pieeja tiek aplūkota 4. nodaļā, kur tiek aprakstīti izstrādātie testu komplektu ģenerēšanas risinājumi.

2.5. Otrās nodaļas secinājumi

2.5.1. Analizētas funkciju izsaukumu līmeņa, komunikācijas līmeņa un lietotāja saskarnes līmeņa testēšanas rīku projektējumu īpašības, kas ir jāievēro, izstrādājot jaunus rīkus. Jaunu rīku izstrāde var būt nepieciešama specifiskos gadījumos, taču esošās rīku projektēšanas pieredzes izmantošana atvieglo izstrādes procesu.

2.5.2. Izpētīta tipiskā testu programmatūras struktūra, tās projektēšanas process un aplūkotas dažas heiristikas. Pieturēšanās pie tipiskās testu programmatūras struktūras atvieglo testu automatizētāju darbu, ievieš vienotu terminoloģiju, atvieglo komunikāciju starp izstrādātāju, testētāju un testu automatizētāju grupām.

2.5.3. Analizēti divi testu automatizācijas procesu modeļi — Fewster un Graham automatizētās testēšanas procesa modelis un Dustin automatizētā testa dzīves cikla metodoloģija. Abi modeļi labi atspoguļo automatizētā testa dzīves ciklu, taču neņem vērā testpiemēru automatizētās ģenerēšanas iespējamību un ir vērsti tikai uz automatizētu izpildi.

2.5.4. Izstrādāts vienots automatizētās testēšanas modelis, kurš apvieno testpiemēru automatizētu ģenerēšanu un automatizētu izpildi vienā procesā. Mūsdienas ir maz izplatīti rīki, kas iekļauj abas šīs iespējas, tāpēc šādas darbību secības nodrošināšanai parasti ir nepieciešama dažādu rīku integrācija. Modelis parāda saiknes starp automatizētās testēšanas aktivitātēm, ļaujot konkrētās situācijās novērtēt dažādu aktivitāšu automatizējamību. Aprakstītas arī izstrādātā modeļa pielietošanas iespējas.

3. AUTOMATIZĒTO TESTU METRIKAS UN EFEKTIVITĀTES NOVĒRTĒŠANA

3.1. Testēšanas rezultativitāte un efektivitāte

Testēšanas process ir paredzēts defektu atklāšanai. Tāpēc ir jābūt veidam, kā no potenciāli bezgalīgas iespējamo testu kopas izvēlēties tādu apakškopu, kas atklātu pēc iespējas vairāk defektu, cik ir iespējams testēšanai atvēlētajā laikā.

3.1.1. Testēšanas pamatmetrikas

Izsmeļoša testēšana nav iespējama [100] tāpēc, ka testēšanas laiks un budžets ir ierobežoti. Testu skaitu, ko ir iespējams sagatavot vai izpildīt, var prognozēt, izmantojot formulas [90]:

$$\begin{aligned} S_1 &= T/t_{vid}; \\ S_2 &= B/b_{vid}; \\ S &= \min(S_1, S_2), \end{aligned} \tag{3.1}$$

kur

T — testēšanai atvēlēts laiks cilvēkstundās;

t_{vid} — viena testpiemēra sagatavošanas vai izpildes vidējais laiks;

B — testēšanai atvēlēts budžets;

b_{vid} — viena testpiemēra sagatavošanas vai izpildes vidējais laiks.

Svarīgākais no testēšanas mērķiem ir defektu atklāšana [72]. Tāpēc, lai testēšana būtu efektīva, testi ir jāizvēlas tādā veidā, lai no vienas puses tiem nepieciešamais laiks nepārsniegtu testēšanai atvēlēto laiku, bet no otras puses, lai tie varētu atklāt pēc iespējas vairāk un pēc iespējas smagākus defektus.

Testēšanas rezultātus mēdz raksturot ar dažādām metrikām, kas ir saistītas ar defektu skaitu, piemēram [69]:

- kumulatīvs atklāto defektu skaits;
- defektu blīvums;

– testēšanas laikā atklāto defektu daļa.

Pēdējo metriku bieži izmanto testēšanas rezultativitātes novērtēšanai. Hutcheson [57] definē testēšanas rezultativitāti (*test effectiveness*) kā testu kopas defektu atklāšanas spējas mēru un piedāvā attiecīgu formulu:

$$E_R = \frac{D_{test}}{D_{test} + D_{liet}} \times 100\%, \tag{3.2}$$

kur

E_R — testēšanas rezultativitāte;

D_{test} — testēšanas laikā atklātu defektu skaits;

D_{liet} — defektu skaits, ko lietotāji atklāj pēc testēšanas.

Formulai (3.2) ir divi trūkumi:

1) tā neņem vērā defektu smagumus;
2) tā ir izmantojama tikai jau notikušā fakta konstatācijai.

Defektu skaits pats par sevi nepietiekami labi raksturo programmatūras kvalitāti [57]. Ja defektu skaits būtu pašmērķis, testētāji varētu koncentrēties uz mazsvarīgu defektu meklēšanas, tādā veidā panākot labākus rādītājus, taču maz ieguldot produkta kvalitātes uzlabošanā. Tāpēc ir noderīgi ņemt vērā defektu smagumus atkarībā no to svarīguma.

Defektu d_i smagumu $S(d_i)$ vērtību skalas var būt dažādas. Viennozīmīguma labad pieņemsim, ka vērtības ir intervālā no 0 (neieskaitot) līdz 1 (ieskaitot), t.i., $0 < S(d_i) \leq 1$.

Pieņemsim arī, ka $S(d_i)$ vērtību skala ir lineāra, t.i., divreiz smagākam defektam atbilst divreiz lielākā vērtība — piemēram, ja ir trīs defekti d_1, d_2 un d_3 ar smagumiem $S(d_1) = 0.8$, $S(d_2) = 0.5$, un $S(d_3) = 0.3$, tad viena d_1 defekta izlabošana uzlabos produkta kvalitāti tādā pašā mērā kā abu defektu d_2 un d_3 izlabošana. Var izskatīt arī speciālu gadījumu, kad smagums ir nulle: $S(d_i) = 0$, un kas nozīmētu, ka d_i par īstu defektu nav uzskatāms.

Ja formulā (3.2) ņem vērā defektu smagumus, tā transformējas par formulu:

$$E_R = \frac{\sum_{d \in \mathcal{D}_{test}} S(d)}{\sum_{d \in \mathcal{D}_{test}} S(d) + \sum_{d \in \mathcal{D}_{liet}} S(d)}, \tag{3.3}$$

kur

E_R — testēšanas rezultativitāte;

$\mathcal{D}_{test}$ — testēšanas laikā atklātu defektu kopa;

$\mathcal{D}_{test}$ — defektu kopa, ko lietotāji aklāj pēc testēšanas;

$S(d)$ — defekta d smagums.

Ievērojot to, ka pie atzīmētajiem pieņēmumiem, no kvalitātes viedokļa divas defektu kopas, kuru smagumu summas ir vienādas, vienādi ietekmē produkta kvalitāti, ir noderīgi ieviest defektpunkta jēdzienu. Par defektpunktu sauksim defekta smaguma mērvienību. Līdz ar to uzskatīsim, ka defektu smagumi vai defektu kopu smagumu summas ir izsakāmas defektpunktos. Tātad defektpunktu skaits ir defektu skaits, svērts ar šo defektu smagumiem. Ieviešot šādu mērvienību, formula (3.3) pēc sava izskata pārverošās formulā (3.2) ar to atšķirību, ka D tajā apzīmē nevis defektu, bet defektpunktu skaitu.

3.1.2. Testēšanas efektivitāte

Testēšanas efektivitātei angļu literatūrā atbilst divi termini — *testing effectiveness* un *testing efficiency*. Bieži tiek izteikti viedokļi par to, ka tie divi termini ir dažādi, taču to skaidrojumos nav vienotības. Katrā gadījumā visbiežāk par atšķirošu faktoru pieņem testēšanas izmaksu dimensiju patērēta laika un budžeta nozīmē. Ja *testing effectiveness* vairāk attiecas uz gala rezultātu un parāda, cik labi testēšana sasniedza tās mērķus, tad *testing efficiency* parāda, kādā mērā sasniegtie rezultāti attaisnoja ieguldīto darbu. Šo apsvērumu dēļ darba ietvaros formulā (3.2) definēto *testing effectiveness* sauksim par testēšanas rezultativitāti, bet testēšanas efektivitāti (*testing efficiency*) balstīsim uz Pfleeger [101] definīciju, ko var izteikt formulas veidā:

$$E = D/T, \tag{3.4}$$

kur

E — testēšanas efektivitāte;

D — testēšanas laikā atklātu defektu skaits;

T — testēšanai patērētais laiks cilvēkstundās.

Tāpat kā iepriekš ar D ir izdevīgi saprast nevis defektu, bet defektpunktu skaitu.

Formulā (3.4) paliek trūkums, ka efektivitāti var izskaitļot tikai tad, kad testēšana jau ir notikusi un tās rezultāti ir zināmi. No šī viedokļa raugoties, tā formula nav lietojama, lai novērtētu testu kopas efektivitāti pirms izpildes. Bet no otrās puses testēšanai patērētu laiku ir

iespējams pietiekami precīzi prognozēt uz priekšu, bet defektpunktu skaita prognozēšanai var izmantot defektu riskus. Šajā nodaļā tiek piedāvāts matemātiskais modelis, kas ļauj prognozēt testēšanas efektivitāti balstoties uz defektu risku analīzi.

3.2. Testa efektivitātes modelis

3.2.1. Defekta risks

Smaguma īpašība piemīt ne tikai jau atklātiem defektiem, bet arī potenciāliem un prognozējamiem. Defektiem, kas nav atklāti, taču ir iespējami, piemīt vēl viena īpašība — to esamības varbūtība $P(d_i)$. No smaguma un varbūtības īpašībām var izskaitļot tādu svarīgu potenciāla defekta īpašību, kā defekta risks:

$$R(d_i) = S(d_i) \cdot P(d_i), \tag{3.5}$$

kur

$R(d_i)$ — defekta d_i risks;

$S(d_i)$ — defekta d_i smagums;

$P(d_i)$ — defekta d_i esamības varbūtība.

Uz riskiem balstītajā testēšanā [5] riski ir galvenais faktors, kas nosaka, kādi testi tiks sagatavoti un izpildīti. Tālāk šajā nodaļā tiek aplūkotas defektu risku un testu efektivitātes savstarpējās attiecības, un kā panākt racionālāku testu izvēli, kas ļaus atklāt defektus ar lielāku varbūtību.

3.2.2. Testa svarīgums

Katram testam τ ir tāds potenciāls defekts vai defektu kopa, kuru esamību vai neesamību tas tests ir spējīgs konstatēt. Tā kā testēšanas mērķis ir defektu atklāšana, testiem, kas nav spējīgi atklāt nevienu defektu, nav jēgas. No otras puses, viena mērķa princips testu projektēšanā, nosaka, ka testam ir jābūt pēc iespējas konkrētam, no kā seko, ka konstatējamo defektu kopa nevar būt liela. Šādu kopu sauksim par testa τ mērķa defektu kopu un apzīmēsim $\mathcal{D}(\tau)$.

Jo vairāk defektu spēj konstatēt tests τ, jo labāks ir tests. No otrās puses, viena smaga defekta konstatācija var būt ievērojami svarīgāka, nekā daudzu mazsvarīgu defektu atklāšana. Tātad testa svarīguma $\vartheta(\tau)$ novērtēšanai būtu jāņem vērā kopas $\mathcal{D}(\tau)$ defektu smagumus defektpunktos. Jāņem vērā arī tas, ka labāks tests ir tests, kas atklāj defektu ar lielāku varbūtību. Kombinējot defektu smagumus un to esamības varbūtību, var nonākt pie formulas:

$$\vartheta(\tau) = \sum_{d_i \in \mathcal{D}(\tau)} S(d_i) \cdot P(d_i) = \sum_{d_i \in \mathcal{D}(\tau)} R(d_i), \tag{3.6}$$

kur

$S(d_i)$ — defekta d_i smagums;

$P(d_i)$ — defekta d_i esamības varbūtība;

$R(d_i)$ — defekta d_i risks.

Pēc būtības šī formula rēķina sagaidāmo atklāto defektpunktu skaita matemātisko cerību.

3.2.3. Testam veltīti laiki

Katram testam τ ir noteiktas laika izmaksas. Ieviesīsim šādus apzīmējumus:

- $T_P(\tau)$ — testa τ sagatavošanas (*preparation*) laiks;
- $T_E(\tau)$ — testa τ izpildes (*execution*) laiks;
- $T_U(\tau)$ — testa τ atjaunošanas (*update*) laiks;
- $T_T(\tau)$ — kopējais (*total*) testam τ veltīts laiks.

Šeit un turpmāk šajā darba nodaļā laika (darbietilpības) vērtības ir izsakāmas cilvēkstundās. Testa atjaunošanas laiks ir laiks, kas veltīts testa uzturēšanai aktuālajā stāvoklī. Tā kā mūsdienās programmatūras izstrāde parasti notiek iteratīvi [77], viens un tas pats tests bieži ir jāatkārto katrā iterācijā, lai nodrošinātu, ka netiek sabojāta jau notestēta funkcionalitāte. Katrā projekta iterācijā var notikt vai jaunu testu sagatavošana vai arī esošu testu atjaunošana, kas ir nepieciešama, lai panāktu testa atbilstību pēdējai testējamās programmas versijai. Šis rādītājs vienam testam var mainīties atkarībā no iterācijā ieviestajām izmaiņām un no tā, kāda ir testa τ testējamā funkcionalitāte. Taču parasti katram testam var uz priekšu prognozēt vidējo laiku, kas būs nepieciešams tā atjaunošanai katrā iterācijā.

Minētās testa īpašības var apvienot formulā (šeit un turpmāk parametrs τ tiek izlaists lasāmības nolūkā):

$$T_T = T_P + n\,T_E + (n-1)\,T_U\,, \tag{3.7}$$

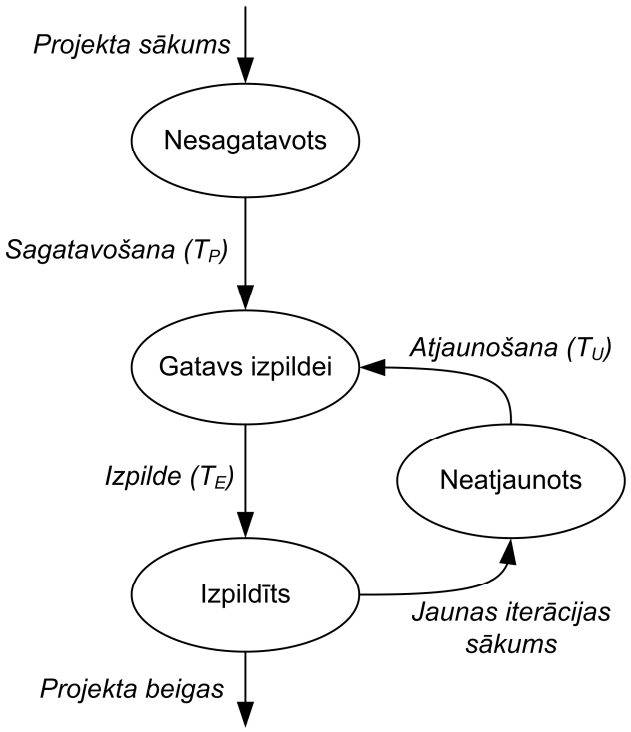
kur

n — testa τ izpilžu skaits.

Tātad katrā projekta brīdī tests τ var būt vienā no stāvokļiem:

- nesagatavots — tests ir apzināts kā iespēja, bet tā sagatavošana vēl netika veikta;
- neatjaunots — tests ir sagatavots kādā no iepriekšējām iterācijām, bet tam ir nepieciešama atjaunošana, lai tas atbilstu aktuālai testējamā produkta versijai;
- gatavs izpildei — tests dotajā iterācijā ir sagatavots, vai atjaunots;
- izpildīts.

Šo stāvokļu izmaiņas ilustrē stāvokļu diagramma 3.1. attēlā.



3.1. att. Testa stāvokļi

Apsverot testu automatizācijas iespēju, parādās papildus laika īpašības:

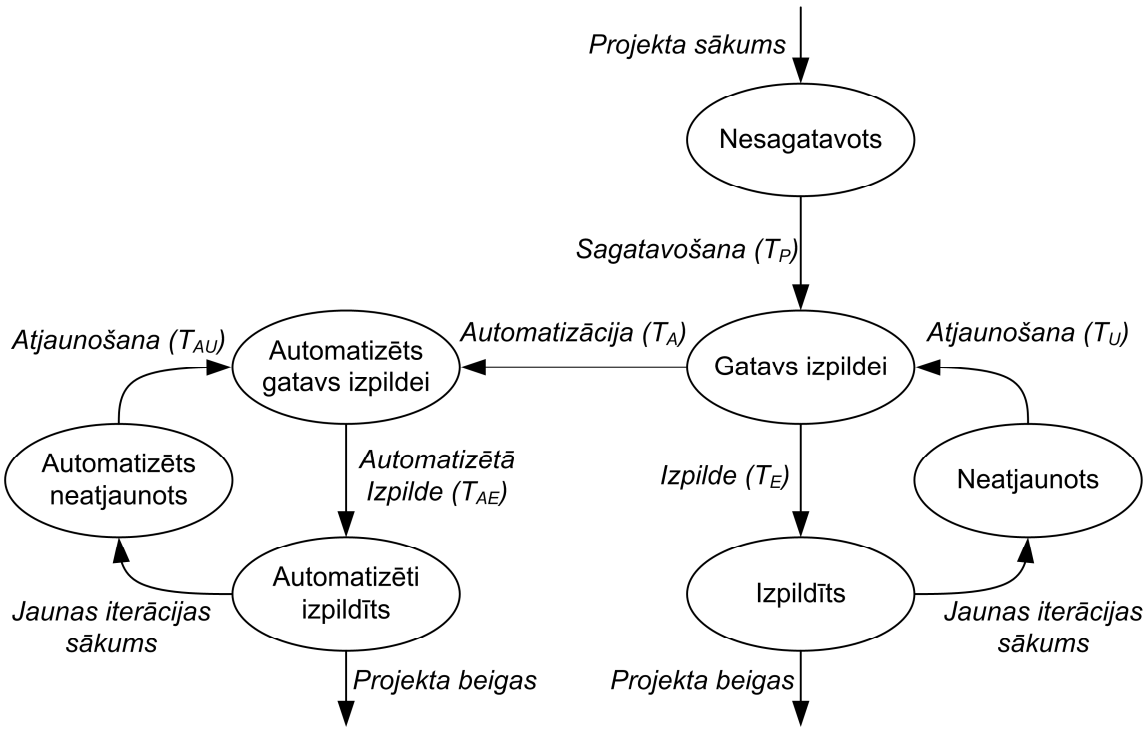
- T_A — testa automatizācijas laiks;
- T_{AE} — automatizētā testa izpildes laiks;

– T_{AU} — automatizētā testa atjaunošanas laiks.

Ja visi testi tiek automatizēti, stāvokļu diagramma būtu līdzīga 3.1. attēlā esošajai ar to atšķirību, ka testa sagatavošana aizņemtu laiku $T_P + T_A$, jo tests sākumā ir jāsagatavo un jāautomatizē, testa izpilde aizņemtu laiku T_{AE}, bet testa atjaunošana — laiku T_{AU}. Taču praksē parasti tiek izmantota hibrīda stratēģija, kurā sākumā testi ir manuāli, un tikai pēc tam tiek automatizēti. Tad jau minētajiem testa stāvokļiem pievienojas jauni:

– automatizēts gatavs izpildei;
– automatizēts neatjaunots;
– automatizēti izpildīts.

Testa stāvokļu diagramma šādā hibrīdajā stratēģijā ir ilustrēta 3.2. attēlā.



3.2. att. Automatizējamā testa stāvokļi

Abas diagrammas ir izveidotas, pieņemot, ka testēšana tiek veikta racionāli, tāpēc tajās iztrūkst pārejas, kas ir iespējamas, bet nav racionālas. Piemēram, testa automatizāciju ir iespējams veikt gan no stāvokļa „gatavs izpildei“, gan no „izpildīts“. Taču ja iterācijā ir plānots testu automatizēt, ir racionālāk sākumā automatizēt testu un pēc tam automatizēti izpildīt, nekā sākumā izpildīt manuāli un pēc tam automatizēt, jo automatizētā testa izpilde prasa ievērojami mazāku laiku. Šī iemesla dēļ 3.2. attēlā automatizācija tiek veikta, kad tests ir stāvoklī „gatavs izpildei“.

Tā kā testa automatizācija ir darbietilpīgs process, pateicoties kuram testa izpilde kļūst ievērojami ātrāka, testu, kas jau ir automatizēts, nav racionāli atkal pārvērst par manuālo. Šī iemesla dēļ 3.2. attēlā testa automatizācija dala stāvokļus divās grupās — pirms automatizācijas un pēc automatizācijas, pie tam no automatizētajiem stāvokļiem vairs nav pārēju atpakaļ.

Jaunas iterācijas sākumā tests pāriet stāvoklī „neatjaunots“ vai „automatizēts neatjaunots“, jo pirms testa izpildes ar jaunu testējamā produkta versiju ir vismaz jāpārliecinās par to, ka tests joprojām ir aktuāls. Tas savukārt prasa laiku, ko var attiecināt uz testa atjaunošanu. Iterācijas beigās testam ir jābūt izpildītam, manuāli vai automatizēti, jo savādāk neattaisnotos testam patērētais laiks.

Šajā hibrīdajā stratēģijā kopējo automatizētajam testam veltītu laiku var novērtēt ar formulu:

$$T_T = T_P + mT_E + mT_U + T_A + (n-m)T_{AE} + (n-m-1)T_{AU}\,, \tag{3.8}$$

kur

n — kopējais testa izpilžu skaits;

m — testa manuālo izpilžu skaits.

Šo modeli var būtiski vienkāršot balstoties uz praktiskiem apsvērumiem.

1. Automatizētā testa izpilde neprasa cilvēka laiku, līdz ar to var uzskatīt, ka attiecīga darbietilpība ir vienāda ar nulli:

$$T_{AE} = 0. \tag{3.9}$$

2. Automatizētā testa sagatavošana un atjaunošana ir sarežģītāka par tā paša testa sagatavošanu un atjaunošanu manuālajā variantā. Šī sarežģītība nosaka attiecīgo darbietilpību palielināšanu automatizācijas gadījumā, ko ir lietderīgi izteikt kā automatizācijas sarežģītības koeficientu α (ievērojot, ka automatizētā testa sagatavošanas laiks ietver testa sagatavošanu kā tādu un tā automatizāciju):

$$\alpha = \frac{T_P + T_A}{T_P} = \frac{T_{AU}}{T_U}\,. \tag{3.10}$$

No kā izriet

$$T_A = (\alpha - 1)T_P;$$
$$T_{AU} = \alpha T_U\ , \tag{3.11}$$

kur

$\alpha > 1$ — automatizācijas sarežģītības koeficients.

Izmantojot formulās (3.9) un (3.11) ieviestos vienkāršojumus, formula (3.8) transformējas par:

$$T_T = \alpha T_P + mT_E + (\alpha(n - m - 1) + m)T_U\ , \tag{3.12}$$

kur

n — kopējais testa izpilžu skaits;

m — testa manuālo izpilžu skaits.

Koeficienta α vērtība ir atkarīga no vairākiem faktoriem, tādiem kā:

- testējamās sistēmas īpašības — cik lielā mērā sistēma un tajā izmantotās tehnoloģijas nodrošina automatizēto testējamību;
- automatizācijas rīks — jo labāk rīks ir piemērots konkrētiem testēšanas uzdevumiem, jo ātrāk ir iespējams veikt automatizēšanu;
- testēšanas komandas pieredze — ja testēšanas komandai nav lielas pieredzes testu automatizācijā, tas var ieviest papildus grūtības;
- testēšanas process — ja testēšanas darbplūsma nav adaptēta testu automatizācijai, tad automatizācijas laiks var palielināties;
- un vairāki citi faktori.

Tā kā faktoru ir daudz, automatizācijas sarežģītības koeficienta α noteikšana ir jābalsta pirmkārt uz iepriekšējo pieredzi līdzīgos projektos, vai tā paša projekta iepriekšējos posmos. Ja testēšanas stratēģija organizācijā ir balstīta uz automatizāciju, projektu nobeiguma stadijā (*post mortem*) ir būtiski novērtēt faktisko automatizācijas sarežģītību, lai iegūtā pieredze būtu izmantojama nākotnē.

Testam τ veltītais laiks vienā i-tajā iterācijā ir atkarīgs no tā, kāds stāvoklis ir testam iterācijas sākumā un vai testu šajā iterācijā ir paredzēts automatizēt. Šo i-tajā iterācijā testam τ veltīto laiku $T_{T\,i}(\tau)$ aprēķinu formulas ir apkopotas 3.1. tabulā (ievērojot ieviestos vienkāršojumus).

3.1. tabula

Testam patērētais laiks $T_{T\,i}$ vienā iterācijā

Stāvoklis iterācijas sākumā	**Stāvoklis iterācijas beigās**	
	Izpildīts	Automatizēti izpildīts
Nesagatavots	$T_P + T_E$	αT_P
Neatjaunots	$T_U + T_E$	$T_U + (\alpha - 1)T_P$
Automatizēts neatjaunots	—	αT_U

3.2.4. Testa efektivitāte

Zinot, kāds laiks ir jāvelta testam τ i-tajā iterācija, un testa svarīgumu, ir iespējams izskaitļot prognozējamo testa efektivitāti $E_i(\tau)$ šajā iterācijā. Balstoties uz formulu (3.4) un reducējot to uz viena testa gadījumu konkrētajā iterācijā, ir jāaizvieto defektu skaits ar testa svarīgumu (sagaidāmo defektpunktu skaitu, ko atklās tests) un par laiku jāpieņem testam veltīts laiks i-tajā iterācija. Rezultātā tiek iegūta formula:

$$E_i(\tau) = \frac{\vartheta_i(\tau)}{T_{T\,i}(\tau)}\ , \tag{3.13}$$

kur

$E_i(\tau)$ — testa τ efektivitāte i-tajā iterācija;

$\vartheta_i(\tau)$ — testa τ svarīgums i-tajā iterācija;

$T_{T\,i}(\tau)$ — testam τ patērētais laiks i-tajā iterācija.

Ievērojot formulu (3.7), kumulatīvā testa efektivitāte n pirmajās iterācijās manuālās testēšanas gadījumā (E_n^{man}) ir rēķināma ar formulu :

$$E_n^{\text{man}} = \frac{\sum_{i=1}^{n} \vartheta_i}{T_P + nT_E + (n-1)T_U}, \tag{3.14}$$

kur

n — testa izpilžu skaits.

Ievērojot formulu (3.12), kumulatīvā testa efektivitāte n pirmajās iterācijās hibrīdās testēšanas gadījumā ($E_{n,m}^{\text{hyb}}$) ir rēķināma ar formulu:

$$E_{n,m}^{\text{hyb}} = \frac{\sum_{i=1}^{n} \vartheta_i}{\alpha T_P + m T_E + (\alpha(n-m-1)+m) T_U}, \tag{3.15}$$

kur

n — kopējais testa izpilžu skaits;

m — testa manuālo izpilžu skaits.

Speciāls $E_{n,m}^{\text{hyb}}$ gadījums, kad tests tiek automatizēts no paša sākumā, t.i., $m = 0$, tiks apzīmēts ar E_n^{auto}.

Tagad ir iespējams izrēķināt, kādā mērā hibrīdās testēšanas stratēģija ir efektīvāka (vai neefektīvāka) par tīri manuālo. Šī attiecība tiks apzīmēta ar $\mathcal{A}^{\text{hyb}}$:

$$\mathcal{A}_n^{\text{hyb}} = \frac{E_{n,m}^{\text{hyb}}}{E_n^{\text{man}}} = \frac{T_P + n T_E + (n-1) T_U}{\alpha T_P + m T_E + (\alpha(n-m-1)+m) T_U}. \tag{3.16}$$

Tīri automatizētās testēšanas gadījumā (kas ir hibrīdās stratēģijas triviāls gadījums, pie $m = 0$), tā attiecība ir vienkāršāka. Šo attiecību sauksim par testa automatizācijas relatīvo efektivitāti:

$$\mathcal{A}_n = \frac{E_n^{\text{auto}}}{E_n^{\text{man}}} = \frac{T_P + n T_E + (n-1) T_U}{\alpha T_P + \alpha(n-1) T_U} = \frac{1}{\alpha}\left(1 + \frac{n T_E}{T_P + (n-1) T_U}\right). \tag{3.17}$$

$\mathcal{A}_n$ vērtība, kas ir lielāka par 1, liecina par to, ka n pirmajās iterācijās automatizētais tests sasniedz augstāku efektivitāti, nekā tāds pats manuālais tests. Un otrādi, ja $\mathcal{A}_n$ ir mazāka par 1, šāda testa automatizācija būtu bijusi neefektīva, vismaz runājot par pirmajām n iterācijām. Praksē var būt vērtīgi novērtēt, pie kāda iterāciju skaita n testa automatizācijas relatīvā efektivitāte $\mathcal{A}_n$ kļūst lielāka par 1, t.i., pēc kuras iterācijas testa automatizācija sāks atmaksāties. Balstoties uz formulu (3.17), risinot nevienādojumu $\mathcal{A}_n > 1$, var konstatēt, ka tāds n pastāv tikai ja izpildās nosacījums:

$$\alpha T_U < T_U + T_E. \tag{3.18}$$

Nosacījumu (3.18) ir viegli interpretēt tā, ka automatizētā testa atjaunošanas laikam ir jābūt mazākam par neautomatizētā testa atjaunošanas un izpildes laiku summu. Savādāk testa automatizācija nedod nekādu priekšrocību, salīdzinot ar manuālo testu. Ja šis nosacījums izpildās, ir iespējams izrēķināt iterāciju skaitu n_A, pie kura testa automatizācija sāk atmaksāties:

$$n_A = \left\lceil \frac{(\alpha - 1)(T_P - T_U)}{T_E - (\alpha - 1)T_U} \right\rceil. \tag{3.19}$$

Ja testa sagatavošanas laiku T_P pieņemt par nosacītu laika vienību, un izteikt T_E un T_U kā daļu no T_P, šajā formulā paliktu tikai trīs parametri. 3.2. tabulā ir apkopotas aprēķinātās n_A vērtības pie dažādiem α, T_U un T_E. Svītra tabulā nozīmē, ka dotajā gadījumā automatizācija neatmaksājas.

3.2. tabula

Iterāciju skaits, pie kura testa automatizācija sāk atmaksāties, atkarībā no α, T_U un T_E

α	1.5			2			4		
T_U	0.001	0.01	0.1	0.001	0.01	0.1	0.001	0.01	0.1
T_E = 0.01	53	99	—	111	—	—	429	—	—
T_E = 0.02	26	33	—	53	99	—	177	—	—
T_E = 0.05	11	11	—	21	25	—	64	149	—
T_E = 0.10	6	6	9	11	11	—	31	43	—
T_E = 0.20	3	3	3	6	6	9	16	18	—
T_E = 0.50	1	1	1	3	3	3	7	7	14
T_E = 1.00	1	1	1	1	1	1	4	4	4

No 3.2. tabulas seko, ka automatizācijas lietderība lielā mērā ir atkarīga no apskatītajiem faktoriem. Ir pietiekoši daudz gadījumu, kad testu nav vērts automatizēt vispār. Piemēram, pie automatizācijas sarežģītības $\alpha = 4$, t.i., kad automatizētā testa izveide prasa četras reizes vairāk laika, nekā līdzvērtīgā manuālā testa izveide, ir lietderīga tikai ja manuālās testa izpildes laiks ir samērojams ar testa sagatavošanas laiku. Testam, kura atjaunošanas laiks ir liels (piemēram, tabulā 10% no sagatavošanai nepieciešamā laika), kas ir tipiski projektam,

kurā notiek straujas izmaiņas, automatizācijas lietderīgums arī nav liels. Labākie kandidāti uz automatizāciju ir tādi testi, kuriem ir neliels atjaunošanas laiks un liels izpildes laiks (piemēram, darbietilpīgi regresa testi), kuriem automatizācija sāk atmaksāties jau pirmajās iterācijās.

3.3. Testu kopas izvēles algoritms

3.3.1. Testu kopas efektivitāte

Ja ir divi testi τ_1 un τ_2, to mērķa defektu kopas var krustoties, piemēram, $D(\tau_1) = \{d_1, d_2, d_3\}$ un $D(\tau_2) = \{d_4, d_5, d_3\}$. Tādā gadījumā testu kopai $\omega = \{\tau_1, \tau_2\}$ defektu kopa ir $D(\omega) = D(\tau_1) \cup D(\tau_2) = \{d_1, d_2, d_3, d_4, d_5\}$. Rēķinot testu kopas ω svarīgumu $\vartheta(\omega)$, tas sanāks mazāks, nekā tajā esošu testu svarīgumu summa. Tā tas ir tādēļ, ka defekts d_3 pieder abu testu mērķa defektu kopām, un tāpēc ja viens no testiem ir izpildīts, otrs jau neko nedos tā defekta konstatēšanas labā, jo to darbu jau ir paveicis pirmais tests.

Tātad testu kopas ω svarīgumu $\vartheta(\omega)$ var izrēķināt pēc formulas:

$$\vartheta(\omega) = \sum_{d_i \in \mathcal{D}(\omega)} R(d_i)\,, \tag{3.20}$$

kur

$\mathcal{D}(\omega) = \bigcup_{\tau_i \in \omega} \mathcal{D}(\tau_i)$ — testu kopas ω mērķa defektu kopa.

Gadījumā, ja testu kopai piederošo testu mērķa defektu kopas nekrustojas (nevienam testam testu kopā nav tāda mērķa defekta, ko varētu atklāt arī cits tests tajā pašā kopā), testu kopas svarīgums būs vienāds ar tai piederošu testu svarīgumu summu:

$$\vartheta(\omega) = \sum_{\tau_i \in \omega} \vartheta(\tau_i)\,. \tag{3.21}$$

Formulu (3.21) ir izdevīgāk izmantot praksē, jo parasti potenciālam defektam vai dažu defektu kopai projektē atsevišķu testu, kura nolūks ir tieši šo defektu atklāšana. Tas neizslēdz sagadīšanos, ka kādu no tiem defektiem spēs atklāt arī kāds cits tests, taču šo aspektu neņem vērā, jo tas būtiski sarežģītu analīzes procesu.

Testu kopas izpildei nepieciešamais laiks i-tajā iterācijā (ieskaitot arī testu sagatavošanas un atjaunošanas darbus) ir rēķināms kā atsevišķiem testiem veltītu laiku summa:

$$T_{T\,i}(\omega) = \sum_{\tau_j \in \omega} T_{T\,i}(\tau_j)\,. \tag{3.22}$$

Testu kopas ω efektivitāti i-tajā iterācijā tādā gadījumā var izrēķināt pēc analoģijas ar formulā (3.13) atspoguļotu atsevišķa testa efektivitāti:

$$E_i(\omega) = \frac{\vartheta_i(\omega)}{T_{T\,i}(\omega)}, \tag{3.23}$$

kur

$E_i(\omega)$ — testu kopas ω efektivitāte i-tajā iterācija;

$\vartheta_i(\omega)$ — testu kopas ω svarīgums i-tajā iterācija;

$T_{T\,i}(\omega)$ — testu kopai ω patērētais laiks i-tajā iterācija.

3.3.2. Testu kopas izvēle

Katrā testēšanas iterācijā ir aktuāls jautājums par to, kā izvēlēties veicamās darbības: kādi testi ir jāsagatavo, jāautomatizē, jāatjauno un jāizpilda. No visu apzinātu testu kopas Ω ir jāizvēlas tāda apakškopa $\omega \subseteq \Omega$, lai tās efektivitāte $E_i(\omega)$ dotajā iterācijā būtu pēc iespējas augstāka un kopējais patērētais laiks $T_{T\,i}(\omega)$ nepārsniegtu iterācijā pieejamu laiku T. Vispārīgā gadījumā šis uzdevums ir ekvivalents klasiskajai mugursomas problēmai [27], kura pieder pie NP-pilno problēmu klases, un nav atrisināma polinomiālajā laikā. Ja visu apzinātu testu kopa Ω ir liela (kas ir visai izplatīts gadījums), vislabākās testu kopas izvēle var izrādīties nepiemērots mērķis. Taču ja iterācijā pieejamais laiks T ir ievērojami lielāks par atsevišķiem testiem τ veltāmajiem laikiem $T_{T\,i}(\tau)$, vislabākās testu kopas izvēle ir arī neracionāls mērķis, jo palielinās laika novērtējumu kļūdu nozīme. Tāpēc praktiskāk ir risināt šo uzdevumu kā nepārtraukto mugursomas problēmu ar alkatīgu algoritmu (*greedy algorithm*).

1. Sākumā izvēlēto testu kopa ir tukša $\omega = \emptyset$, bet iterācijā atlikušais laiks $T_{\text{atl}} = T$.

2. Katram testam τ no iterācijā vēl nepaņemtiem testiem un kuriem $T_{T\,i}(\tau) < T_{\text{atl}}$, jāizrēķina efektivitāte $E_i(\tau)$, izmantojot datus par testa τ svarīgumu un tam nepieciešamo laiku (pēc 3.1. tabulas), apsverot automatizācijas iespēju.
3. Jāizvēlas tests τ_{ef}, kuram efektivitāte izrādījusies visaugstākā, jāpievieno τ_{ef} kopai ω, un iterācijā atlikušais laiks jāsamazina par testam τ_{ef} nepieciešamo laiku: $T_{\text{atl}} \coloneqq T_{\text{atl}} - T_{T\,i}(\tau_{\text{ef}})$.
4. Process jāturpina no 2. soļa kamēr paliek nepaņemti testi, kuriem $T_{T\,i}(\tau) < T_{\text{atl}}$.

Šis testu izvēles algoritms nodrošina testu kopas izvēli, kuras efektivitāte ir tuva visaugstākajai vienā konkrētajā iterācijā. Taču ja iterāciju skaits ir paredzams liels, šis algoritms ir pārāk alkatīgs. Piemēram, testa automatizācija reti atmaksājas tajā pašā iterācijā, kad to veic — automatizācijas priekšrocības parādās vēlāk.

Tāpēc var būt lietderīgi izvēlēties testus tā, lai testu kopas efektivitāte būtu pēc iespējas lielāka nevis vienā — tekošajā iterācijā, bet lai būtu pēc iespējas lielāka tās kumulatīvā efektivitāte nākamajās tuvākajās k iterācijās. Tādā gadījumā piedāvātajā algoritmā ir jāmaina 2. solis un katram testam jārēķina nevis efektivitāte $E_i(\tau)$, bet kumulatīvā efektivitāte $E_{i:k}(\tau)$:

$$E_{i:k}(\tau) = \frac{\sum_{j=i}^{i+k-1} \vartheta_j(\tau)}{\sum_{j=i}^{i+k-1} T_{T\,j}(\tau)}\ , \tag{3.24}$$

kur

$E_{i:k}(\tau)$ — testa τ kumulatīvā efektivitāte k iterācijās, sākot ar i-to iterāciju;

$\vartheta_j(\tau)$ — testa τ svarīgums j-tajā iterācija;

$T_{T\,j}(\tau)$ — testam τ patērētais laiks j-tajā iterācija.

Gadījumā, ja $i = 1$, t.i., tests vēl nav sagatavots, formula (3.24) sanāk ekvivalenta formulām (3.14) un (3.15). Pretējā gadījumā, kad tests jau ir sagatavots, formulas (3.24) saucējs ir aprēķināms (izmantojot 3.1. tabulas formulas) kā:

- $k(T_U + T_E)$, ja tests nav un netiek automatizēts,
- $k\alpha T_U$, ja tests jau ir un tātad paliek automatizēts,
- $T_U + (\alpha - 1)T_P + (k-1)\alpha T_U$, ja i-tajā iterācijā sagatavotais tests tiek pārvērsts par automatizēto.

Formulu (3.13), (3.14), (3.15) un (3.24) skaitītāju novērtēšana ir problemātiska. Parasti var pietiekami precīzi novērtēt defektu smagumu un varbūtību tekošajā iterācijā, bet

prognozēt tos nākamajās iterācijās ir grūtāk. Ja defektu smagums parasti paliek nemainīgs, tad to esamības varbūtības var mainīties atkarībā no tekošās iterācijas rezultātiem, kā arī no izmaiņām, kas tiek ieviestas testējamajā produktā. Līdz ar to mainās arī testu svarīgumi. Prakse liecina, ka pēc testa izpildes, tā mērķa defektu kopas defektu esamības varbūtība nākamajā iterācijā samazinās — ja tiek konstatēts, ka defekts ir, tas tiek labots, bet ja defekts nav konstatēts tad var sagaidīt, ka tas netiks konstatēts arī nākamajā iterācijā. Šo novērojumu var modelēt ieviešot svarīguma samazināšanas koeficientu $\beta < 1$: ja testam τ i-tajā iterācijā ir svarīgums $\vartheta_i(\tau)$, un tests šajā iterācija tiks izpildīts, var sagaidīt, ka nākamajā iterācijā tā svarīgums būs $\vartheta_{i+1}(\tau) = \beta\vartheta_i(\tau)$. Tādā gadījumā minēto formulu skaitītājus var novērtēt pēc formulām:

$$\sum_{i=1}^{n} \vartheta_i(\tau) = \sum_{i=1}^{n} \beta^{i-1}\vartheta_1(\tau) = \vartheta_1(\tau)\frac{\beta^n - 1}{\beta - 1};$$
$$\sum_{j=i}^{i+k-1} \vartheta_j(\tau) = \sum_{j=i}^{i+k-1} \beta^{j-i}\vartheta_i(\tau) = \vartheta_i(\tau)\frac{\beta^k - 1}{\beta - 1}. \quad (3.25)$$

3.3.3. Testu kopas izvēles piemērs

Šajā sadaļā tiek aprakstīts vienkāršs testu kopas izvēles algoritma pielietošanas piemērs. Datu apjoms šajā piemērā ir nesalīdzināmi mazāks, nekā tas ir reālos projektos, bet tas ir pietiekams, lai ilustrētu metodes būtību.

Kādā hipotētiskajā projektā testēšanas komandas uzdevums ir veikt programmatūras moduļa baltās kastes testēšanu. Pēc analīzes tika konstatēts, ka modulī var būt 17 potenciāli defekti, un šiem defektiem tika novērtēts to smagums $S(d)$ (cik nopietnas var būt sekas, ja defekts modulī ir) un to esamības varbūtība $P(d)$. Tika nolemts, ka katram no 17 defektiem būtu izveidojams atsevišķs tests. Šajā gadījumā, par cik mērķa defektu kopa sastāv no viena defekta $\mathcal{D}(\tau_i) = \{d_i\}$, testu svarīgums ir vienāds ar attiecīgā defekta risku $\vartheta(\tau_i) = R(d_i)$. Katram testam tika novērtēts darba apjoms, kas ir nepieciešams to sagatavošanai, atjaunošanai un izpildei (T_P, T_U, T_E). Balstoties uz pieredzi iepriekšējos projektos, testēšanas komanda novērtē automatizācijas sarežģītības koeficientu $\alpha = 2$, bet svarīguma samazināšanas koeficientu $\beta = 0.8$. Vienai iterācijai pieejamā darbietilpība $T = 40$ cilvēkstundas.

Šie hipotētiskā projekta dati ir apkopoti 3.3. tabulā. Pie šādas iterācijai pieejamās darbietilpības, ir skaidrs, ka neizdosies sagatavot un izpildīt visus testus uzreiz pirmajā iterācijā.

3.3. tabula

Testu kopas izvēles piemērs — pirmdati

Npk	Defekta tips	$P(d)$	$S(d)$	$R(d), \vartheta(\tau)$	T_P	T_U	T_E
1	Vispārīgā struktūra	0.0096	0.9	0.00864	10	0.50	3.0
2	Vadības plūsmas predikāti	0.0165	0.3	0.00495	7	0.35	2.0
3	Vadības plūsmas loģika	0.0181	0.5	0.00905	8	0.40	2.0
4	Cikla beigu nosacījumi	0.0033	0.3	0.00099	3	0.15	3.5
5	Iterāciju mainīgā apstrāde	0.0001	0.2	0.00002	2	0.10	1.5
6	Citi ciklu un iterāciju defekti	0.0040	0.1	0.00040	7	0.35	4.0
7	Vadības plūsmas stāvokļi	0.0006	0.6	0.00036	7	0.35	3.5
8	Citi vadības plūsmas defekti	0.0760	0.1	0.00760	8	0.40	3.0
9	Algoritmi	0.0075	0.7	0.00525	10	0.50	5.0
10	Izteiksmju aprēķini	0.0275	0.5	0.01375	5	0.25	3.0
11	Aritmētiskās izteiksmes	0.0172	0.3	0.00516	2	0.10	2.0
12	Loģiskās izteiksmes	0.0103	0.3	0.00309	2	0.10	1.0
13	Inicializācija	0.0187	0.2	0.00374	4	0.20	2.0
14	Tīrīšana	0.0006	0.2	0.00012	3	0.15	1.0
15	Precizitāte	0.0054	0.3	0.00162	4	0.20	1.0
16	Izpildes laiks	0.0029	0.2	0.00058	3	0.15	2.0
17	Citi datu apstrādes defekti	0.0611	0.1	0.00611	7	0.35	4.0

3.3. tabulā esošo defektu tipu uzskaitījums un to varbūtības ir balstītas uz B. Beizer grāmatā [16] apkopotu statistiku par dažādu defektu parādīšanas biežumu reālos projektos. Pārējie dati ir patvaļīgi izvēlēti piemēra nolūkos.

Lai 1. iterācijā izvēlētos testu kopu, ir jānovērtē katra testa efektivitāte un laiks, kas jāiegulda. Pieņemsim, ka šajā projektā tika pieņemts lēmums izvēlēties tādu testu kopu, kas tuvākajās 3 iterācijās dotu pēc iespējas lielāku efektivitāti. Tātad ir jāaprēķina E_3^{man} un E_3^{auto} pēc formulām (3.14) un (3.15), ievērojot (3.25), kā arī pirmajā iterācijā veltītu laiku manuālās testēšanas gadījumā (tiks apzīmēts ar T_M) un automatizētās testēšanas gadījumā (tiks apzīmēts ar T_A), izmantojot 3.1. tabulā esošas formulas.

3.4. tabulā šo aprēķinu rezultāti ir apkopoti. Var redzēt, ka visaugstāko efektivitāti 3 iterācijās dod 10. tests tā automatizācijas gadījumā. Tas tiek izvēlēts un iterācijā atlikušais laiks samazinās par $T_A(\tau_{10}) = 10$ cilvēkstundām. Otrais tiek izvēlēts 11. tests (nākamā augstākā efektivitāte), un no iterācijai pieejamā laika tiek atskaitītas vēl $T_A(\tau_{11}) = 4$ cilvēkstundas. Turpinot šo procesu tiek izvēlēti 12. tests (automatizētais) un 3. tests (manuālais). Šajā brīdī paliek pieejamas 12 cilvēkstundas, un kaut arī nākamais pēc efektivitātes būtu 1. tests (manuālais), tam vairs nepietiek laika. Tāpēc tiek izvēlēts nākamais, 8. tests (manuālais) un paliek tikai viena cilvēkstunda laika, kas vairs nav pietiekama nevienam testam.

3.4. tabula

Testu kopas izvēles piemērs — 1. iterācija

Npk	$\vartheta(\tau)$	T_M	T_A	$E_3^{\text{man}} \cdot 10^{-3}$	$E_3^{\text{auto}} \cdot 10^{-3}$	Veids	Laiks
1	0.00864	13	20	1.004	0.878		
2	0.00495	9	14	0.839	0.719		
3	0.00905	**10**	16	**1.416**	1.150	manuālais	10
4	0.00099	6.5	6	0.171	0.336		
5	0.00002	3.5	4	0.007	0.010		
6	0.00040	11	14	0.048	0.058		
7	0.00036	10.5	14	0.046	0.052		
8	0.00760	**11**	16	**0.997**	0.966	manuālais	11
9	0.00525	15	20	0.474	0.534		
10	0.01375	8	**10**	2.237	**2.796**	automatizētais	10
11	0.00516	4	**4**	1.499	**2.623**	automatizētais	4
12	0.00309	3	**4**	1.396	**1.571**	automatizētais	4
13	0.00374	6	8	0.845	0.951		
14	0.00012	4	6	0.044	0.041		
15	0.00162	5	8	0.507	0.412		
16	0.00058	5	6	0.147	0.197		
17	0.00611	11	14	0.731	0.887		
						Kopā	39

Var izrēķināt arī 1. iterācijā sasniedzamu efektivitāti, izrēķinot izvēlēto testu svarīgumu summu un dalot to ar kopējo patērēto laiku. Šajā gadījumā tas sanāk $0.991 \cdot 10^{-3}$ sagaidāmie defektpunkti cilvēkstundā.

Testu izvēle nākamajās iterācijās notiek pēc tā paša principa, izmantojot nākamās iterācijas datus.

3.4. Trešās nodaļas secinājumi

3.4.1. Analizēti testa efektivitāti ietekmējoši faktori — mērķa defektu riski un testu sagatavošanas, atjaunošanas un izpildes laiki manuālo un automatizēto testu gadījumos.

3.4.2. Izstrādāts testa efektivitātes matemātiskais modelis, kas ļauj aprēķināt prognozējamo testu efektivitāti vienā noteiktajā, kā arī vairākās testēšanas procesa iterācijās uz priekšu.

3.4.3. Izstrādāts efektīvās testu kopas izvēles algoritms. Ar tā palīdzību ir iespējams panākt testēšanas laika racionālāku izmantošanu, koncentrējoties uz testiem, kas spēj atklāt defektus ar lielāku varbūtību.

4. AUTOMATIZĒTO TESTU KOMPLEKTU ĢENERĒŠANAS RISINĀJUMA IZSTRĀDE

4.1. Testu kopu izpildes automatizācija

Viena testpiemēra automatizēta izpilde ir relatīvi vienkāršs process. Automatizētais testpiemērs izpildās bez cilvēka līdzdalības — no cilvēka ir nepieciešams tikai palaist testpiemēru uz izpildi. Kad testu kopa satur daudz testpiemēru, katra atsevišķa testpiemēra manuāla palaišana uz izpildi ir neefektīva. Tāpēc ne vien testpiemēru automatizācijas metodes ir svarīgas, svarīgi ir arī atrast metodes lielu testu kopu izpildes automatizācijai. Tas ne vienmēr ir triviāls uzdevums.

Šajā apakšnodaļā tiek izskatītas dažas testu kopu izpildes alternatīvas, kuru izvēle ietekmē to automatizējamību, un tad tiek analizētas vairākas testu dziņu projektēšanas iespējas.

Abstrahējoties no testu kopas iekšējās uzbūves, t.i., no tā, kādā veidā tiek uzdota testpiemēru izpildes secība tajā, var aplūkot divus testu kopu izpildes aspektus: kā testu kopa tiek palaista uz izpildi un kādā veidā tiek prezentēti testu rezultāti. No šo divu aspektu viedokļa testu kopu automatizēto izpildi var klasificēt pēc diviem parametriem, attiecīgi pēc aktivizēšanas tipa un pēc rezultātu tipa.

Izpildes aktivizēšanas tipi

Automatizētā testu kopas izpilde nozīmē to, ka atsevišķi kopas testpiemēri nav jāpalaiž individuāli. Jautājums tad paliek kā tiek palaista visa testu kopa. Šeit ir divas alternatīvas:

- manuālā palaišana, kad cilvēks kaut kādā veidā iniciē testu kopas izpildi;
- automātiskā palaišana, kad testu kopa sāk izpildīties pati bez cilvēka līdzdalības.

Ja atgriezties pie atsevišķiem automatizētajiem testpiemēriem, tad arī var izdalīt divas situācijas. Testpiemērs var būt automatizēts tā, ka cilvēkam tā palaišanai ir nepieciešams izpildīt vienu komandu (vai nospiest pogu), bet var būt arī tā, ka pirms tās komandas izpildes (vai pogas nospiešanas) ir jāveic kaut kādi konfigurācijas darbi. Šie konfigurācijas darbi var būt nepieciešami, lai nodrošinātu testējamās programmas sākuma stāvokli atbilstoši testpiemēra pirmsnosacījumiem. Piemēram, ja testpiemērs ir vērsts uz datu dzēšanas

funkcionalitātes pārbaudi, pirms tā izpildes ir jānodrošina, ka sistēmas datubāzē ir dati, kas ir jāizdzēš.

Testu kopā šo pirmsnosacījumu var būt ievērojami vairāk, jo tajā ir vairāki testpiemēri. Gadījumā ja pirms testu kopas izpildes ir jāveic noteikti konfigurācijas darbi, tikai pirmā alternatīva (manuālā palaišana) ir pieejama. Lai izvairītos no šīs problēmas, nepieciešamo konfigurācijas darbu veikšana arī var būt automatizēta un iekļauta testu kopas skriptos. Ja tā ir, tad visas testu kopas palaišanai patiešām var būt nepieciešams tikai izpildīt vienu komandu vai nospiest pogu, un tādā gadījumā kļūst pieejama arī otrā alternatīva — automātiska palaišana.

Automātiskā testu kopas palaišana nozīmē, ka cilvēks neiniciē testu kopas izpildi, bet tā iniciējas pati saskaņā ar uzdotiem noteikumiem (scenāriju). Šie noteikumi var būt dažādi.

- Testu kopa var periodiski izpildīties ik pēc noteikta laika intervāla. Piemēram, katru nakti, kad neviens darbinieks nestrādā, un galddatoru resursi netiek izmantoti, tos var izmantot automatizētās testēšanas sistēma, izpildot noteiktu testu kopu. Lielākām testu kopām, kurām ir nepieciešams ievērojami vairāk laika, bet kuras notestē programmu daudz detalizētāk, ieplānotais izpildes laiks var būt brīvdienas.
- Testu kopa var izpildīties pie noteiktiem notikumiem. Piemēram, integrāciju serverī, kurā programmētāji ielādē jaunu programmas kodu vai esoša koda modifikācijas, un kur automātiski vai manuāli notiek izstrādājamās programmas būvēšana (kompilācija, asamblēšana), uzreiz pēc tās var automātiski tikt palaista testu kopa.

Jāatzīmē, ka par īsti automatizēto testu kopu izpildi var uzskatīt tikai tādu, kas neprasa cilvēku papildus darbu pirmsnosacījumiem atbilstošas konfigurācijas nodrošināšanai — īsti automatizēto testu kopai tas ir jādara pašai automātiski.

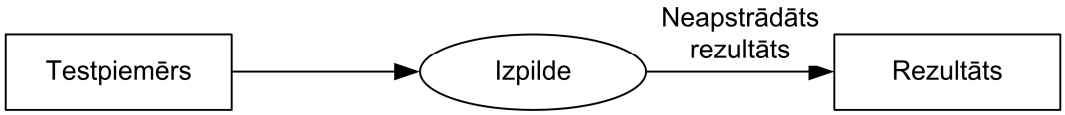
Rezultātu tipi

Automatizētās testu kopas izpildes laikā tiek ģenerēti testu rezultāti. Testējamā programma tās darbināšanas rezultātā izvada noteiktus datus, no kuriem tiek atvasināti testu rezultāti. Pēc tā, kā notiek šī atvasināšana, rezultātus var iedalīt trijos paveidos:

- neapstrādāti rezultāti. Testi neko nedara ar programmas izvadi, un programmas izvade kļūst par testu rezultātiem;

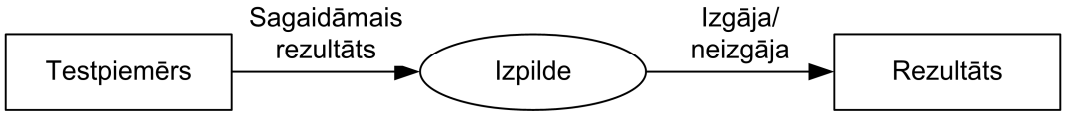
- atbilstība sagaidāmajam rezultātam. Katrs testpiemērs „zina“ kādi rezultāti programmai jāizvada. Iegūtais rezultāts tiek salīdzināts ar sagaidāmo un salīdzināšanas rezultāti kļūst par testu rezultātiem;
- orākula novērtējums. Pašiem testpiemēriem nav zināms sagaidāmais rezultāts, taču automātiskais orākuls var novērtēt iegūto rezultātu korektumu. Novērtēšanas rezultāti kļūst par testu rezultātiem.

Neapstrādāti rezultāti ir primitīvākais rezultātu veids, kas ļauj atlikt līdz vēlākam laikam vērtējumu par to, vai testi ir izgājuši vai nav izgājuši. Testu kopa šajā gadījumā drīzāk kalpo par informācijas savākšanas līdzekli, nevis par tīri testēšanas līdzekli. Testa rezultātos ir redzama programmas izvade uz testpiemēra datiem, kā tas ir ilustrēts 4.1. attēlā. Lēmumu par to, vai testi ir izgājuši un vai programmā ir kļūdas, pieņem cilvēks, kas šos rezultātus izskata un analizē. Tas nozīmē, ka automatizētības pakāpe šai pieejai ir salīdzinoši zema — ir nepieciešams rūpīgs intelektuāls cilvēka darbs, lai nonāktu pie gala slēdziena. Taču šī pieeja ir vienīgā iespējamā, kad nevar identificēt sagaidāmos rezultātus testpiemēru projektēšanas stadijā un nevar arī automatizēt neapstrādātu rezultātu analīzi, proti, izveidot automātisko orākulu.



4.1. att. Neapstrādāti izpildes rezultāti

Otrā paveida rezultātu veids — uzdotie sagaidāmie rezultāti ļauj pašam testa skriptam secināt, vai testpiemērs ir izgājis vai nē. Sagaidāmais rezultāts šajā pieejā ir neatņemama testpiemēra datu sastāvdaļa, t.i. skriptam, kas izpilda testus, tiek padoti dati, kas satur gan ievaddatus (kas tiks padoti testējamās programmas ievadē), gan arī sagaidāmo rezultātu dati. Ja programma testpiemēra izpildes rezultātā izdos rezultātus, kas sakritis ar sagaidāmo, tas ir uzskatāms par izgājušu, savādāk tiek konstatēta programmas kļūda. Tātad šādas testu kopas izpildes rezultātos katram kopas testpiemēram ir redzams tikai rezultāts „izgāja“ (*passed*) vai „neizgāja“ (*failed*). Šis process ir ilustrēts 4.2. attēlā.

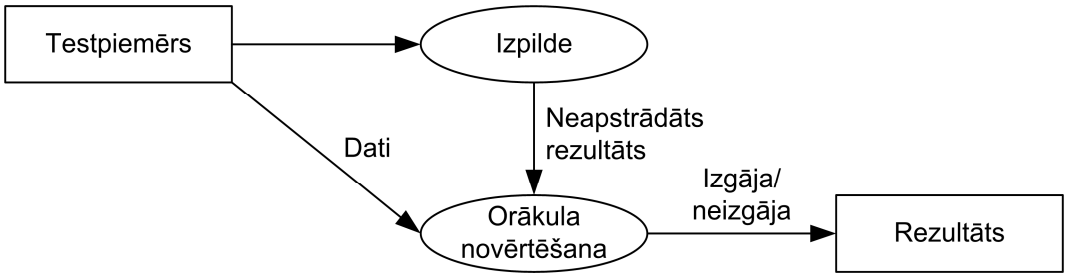


4.2. att. Atbilstība sagaidāmajiem rezultātiem

Vienkāršākajos gadījumos, kad testa skripts sastāv no programmas (vai arī apakšprogrammas) darbināšanas, programma izvada vienīgo rezultātu, skriptam ir jāveic viena vienīgā pārbaude. Sarežģītākajos gadījumos, kad testa skripts sastāv no vairākiem soļiem, kas ir parasta situācija grafiskajām lietotāja saskarnēm, skripts var veikt arī vairākas tādas pārbaudes starp dažādiem soļiem, ar tā saucamo apgalvojumu (*assertion*) jeb kontrolpunktu (*checkpoint*) palīdzību [83]. Tādos gadījumos testpiemēra dati var saturēt vairākus sagaidāmo rezultātu datus dažādiem kontrolpunktiem. Lai iegūtu testa rezultātu, vai tas ir izgājis vai nē, tiek apvienoti vairāku kontrolpunktu izpildes rezultāti — tests ir izgājis tad un tikai tad, ja ir izgājuši visi kontrolpunkti.

Šo pieeju var uzskatīt par vislabāko, jo testu kopas izpildi var automatizēt pilnīgi — gadījumā, ja visi testpiemēri iziet, rezultātā var parādīt tikai šo faktu. Cilvēkam, tātad nav jāveic kaut kāda papildus analīze, lai nonāktu pie šī secinājuma. Otrā pozitīva īpašība šai pieejai ir tāda, ka sagaidāmajiem rezultātiem glabājoties kopā ar ievaddatiem, ekspertiem ir viegli caurskatīt un novērtēt testpiemēru korektumu.

Taču ne vienmēr var precīzi nodefinēt, kādi ir sagaidāmie rezultāti. Piemēram, tas tā var būt, kad programmas izpildē pastāv gadījuma faktors — tad nevar pateikt priekšā, kas konkrēti tiks izvadīts tās korektas darbināšanas rezultātā. Risinājums varētu būt neapstrādātu rezultātu izvadīšana, taču bieži ir tā, ka var noformulēt precīzu algoritmu programmas izdota rezultāta korektuma pārbaudei. Ja šo algoritmu var realizēt automātiski izpildāmas funkcijas veidā, šī funkcija var kalpot par orākulu, kas novērtē neapstrādātus rezultātus un izdod rezultātu formā „izgāja“ vai „neizgāja“, kā tas ir parādīts 4.3. attēlā. Testu orākuli var būt ļoti dažādu veidu, un pastāv vairākas orākulu izstrādes metodes [12].



4.3. att. Orākula novērtējums

Kaut arī orākula izmantošana ir daudz labāka par neapstrādātu rezultātu izvadi, tai ir tāds trūkums, ka sagaidāmie rezultāti nav testpiemēra datu sastāvā un tāpēc testpiemēru kvalitātes novērtēšana ir apgrūtināta — lai pārliecinātos, ka testi strādā pareizi, ir jāpārliecinās par orākula darbības korektumu.

4.2. Testu dziņu veidi

Testu kopa sastāv no testpiemēriem, bet testpiemēru izpildi nodrošina testu skripti. Tātad testu skripts realizē dinamisku testa darbību (testējamās programmas palaišanu, darbināšanu, pārbaužu veikšanu utt.), bet testpiemērs nosaka konkrētus datus jeb parametrus, ar kuriem testa skripts tiek darbināts. Lai izpildītu testu kopu, ir jāizpilda visi tajā esošie testpiemēri, palaižot attiecīgus testu skriptus. Programmatūras vienību, kas veic šo darbību sauc par testu dzini (*test driver*).

Testu dziņus var realizēt vairākos veidos, bet visas testu dziņu realizācijas var iedalīt divās būtiski atšķirīgajās kategorijās — statiskajos un dinamiskajos.

4.2.1. Statiskie testu dziņi

Galvenā statisko testu dziņu raksturīpašība ir tāda, ka testpiemēru izpildes secība tajā ir viennozīmīgi definēta — to nosaka dziņa izstrādātājs. Izskatīsim šīs pieejas divas alternatīvas.

Lineārie dziņi

Vienkāršākā testu dziņa arhitektūra ir lineāras izpildes pieeja. Šajā gadījumā testu dziņa izstrādātājs veido testpiemēru sarakstu $D = (t_1 t_2 \dots t_n)$ un tā izpilde arī notiks pēc šī saraksta.

Šī ir veiksmīgāka pieeja vienkāršām neinteraktīvajām programmām, kuras apstrādā noteiktu ievadi un izdod noteiktu izvadi, gadījumos, kad testpiemēru skaits ir pietiekami mazs, lai būtu pārskatāms. Testpiemēru secībai šajā gadījumā nav nozīmes, jo katrs testpiemērs ir neatkarīgs no pārējiem — visas testpiemēru permutācijas ir ekvivalentas un, veidojot sarakstu, var ņemt vērā tikai to loģisko nozīmi, piemēram, izvietojot „līdzīgus“ testpiemērus viens otram blakus.

Interaktīvajām programmām, kā ari neinteraktīvajām programmām ar pastāvīgu (*persistent*) stāvokli var būt nepieciešams iekļaut šajā secībā papildus darbības, kas nodrošina

testpiemēru pirmsnosacījumus un attīra programmas stāvokli no testpiemēru darbināšanas sekām. Tāda gadījumā testu dziņa modelis kļūst sarežģītāks, jo starp testpiemēriem darbināšanas secībā parādās tās papildus darbības p_i: $D = (p_1 t_1 p_2 t_2 \dots t_n p_{n+1})$.

Šajā gadījumā testpiemēru secībai jau ir būtiska nozīme, jo papildus darbību saturs ir lielā mērā atkarīgs no šīs secības. Tāpēc sarežģītākām testu kopām lineārais modelis var būt nepietiekami efektīvs.

Strukturētie dziņi

Lai mazinātu uzturēšanas problēmas, kas rodas papildus darbību iekļaušanas dēļ, var testu kopu strukturēt, apvienojot „līdzīgus" testpiemērus grupās jeb apakškopās. Šo strukturēšanu var veikt vairākos veidos atkarībā no testpiemēru īpašībām. Izskatīsim dažus testpiemērus.

Pieņemsim, ka noteiktam testu skaitam ir vienādi pirmsnosacījumi, pie tam testpiemēru izpilde programmas stāvokli no pirmsnosacījumu viedokļa neietekmē (t.i. pirmsnosacījumu apgalvojumi paliek spēkā arī pēc šo testpiemēru izpildes). Tad šādus testpiemērus var apvienot grupā $D_k = \left(p_k^\alpha t_{k1} t_{k2} \dots t_{kn_k} p_k^\beta\right)$, kur p_k^α ir k-tās grupas pirmsnosacījumu nodrošināšanas papildus darbība, bet p_k^β — testpiemēru seku attīrīšanas darbība. Tad pilnu testu kopu var projektēt, kā apakškopu secību $D = (p^\alpha D_1 D_2 \dots D_N p^\beta)$, kur p^α un p^β ir papildus darbības, kas attiecās uz visām grupām.

Šāda pieeja dod iespēju pirmkārt samazināt papildus darbību skaitu, otrkārt padarīt atsevišķu grupu testpiemēru secību par viegli modificējamu. Patiešām, katrā atsevišķā grupā testpiemēru secībai nav nozīmes, kā arī nav nozīmes grupu secībai pilnā testu kopā. Tas nozīmē, ka ir ļoti vienkārši iespraust jaunus testus esošajās grupās, kā ari izveidot jaunas grupas.

Mēs apskatījām divu līmeņu testa kopas struktūru — pirmajā līmenī ir testpiemēru grupas, bet otrajā, zemākajā līmenī — atsevišķi testpiemēri. Šo pieeju ir viegli vispārināt uz patvaļīga līmeņu skaitu.

Gadījumos, kad „līdzīgi" testpiemēri maina stāvokli tādā veidā, ka pēc katra atsevišķa testpiemēra izpildes pirmsnosacījumi vairs nav spēkā, ir nepieciešama cita pieeja, proti, pēc katra testpiemēra ir jāveic papildus darbība, kas attīra testpiemēra izpildes sekas, un atkārtoti jānodrošina pirmsnosacījumu izpildīšanos. Taču šādā gadījumā pirms katra grupas testpiemēra ir jāizpilda viena un tā pati pirmsnosacījumus nodrošinoša darbība, un pēc katra

testpiemēra — viena un tā pati stāvokļa attīrīšanas darbība, $D_k = \left(p_k^{\alpha} t_{k1} p_k^{\beta} p_k^{\alpha} t_{k2} p_k^{\beta} \dots p_k^{\alpha} t_{kn_k} p_k^{\beta}\right)$.

Šāda aspekta nodrošināšanu var padarīt par testpiemēru grupas pienākumu, t.i., grupas definēšanas līmenī var pateikt, ka attiecīgas papildus darbības ir jāizpilda pirms un pēc katra testpiemēra: $D_k = \left((\dot{p}_k^{\alpha}) t_{k1} t_{k2} \dots t_{kn_k} (\dot{p}_k^{\beta})\right)$. Dotajā pierakstā punkts virs p apzīmē atkārtošanos pirms un attiecīgi pēc katra testpiemēra. Tieši šī testu komplektu struktūra ir raksturīga xUnit rīkiem [56].

Apvienojot abas pieejas, var nonākt pie kombinētas testpiemēru grupas struktūras:

$$D_k = \left(p_k^{\alpha} (\dot{p}_k^{\alpha}) t_{k1} t_{k2} \dots t_{kn_k} (\dot{p}_k^{\beta}) p_k^{\beta}\right) \tag{4.1}$$

Šajā gadījumā ir gan papildus darbības, kas izpildot grupu būs nepieciešams veikt tikai vienu reizi — sākumā un beigās, gan papildus darbības, kas jāveic pirms un pēc katra testpiemēra. Arī šim modelim uzturamība ir būtiski vienkāršāka nekā tīri lineārajam modelim, kurā ir jārūpējas par katru pāreju starp kaimiņu testpiemēriem. Un arī šis kombinētais modelis ir paplašināms uz patvaļīgu grupu un apakšgrupu līmeņu skaitu.

4.2.2. Dinamiskie testu dziņi

Pat strukturētajos vairāklīmeņu dziņos testpiemēru izpildes secība ir fiksēta, t.i., to specificē dziņa izstrādātājs. Šo secību ir viegli iegūt rekursīvi izvēršot atsevišķas grupas, jo katrā līmenī ir fiksēta gan grupu secība, gan testpiemēru secība grupās. Dinamisko testu dziņu raksturīpašība ir tāda, ka tajos secība netiek uzdota tiešā veidā. Tā vietā tiek deklarēti noteikti testpiemēru izpildes nosacījumi un dziņa automātiskās plānošanas algoritms izveido testpiemēru secību automātiski no šiem nosacījumiem.

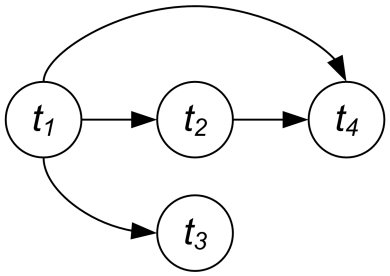
Pastāv vairāki šāda veida plānotāji [39], katrs orientēts uz noteiktas īpašas situācijas apstrādi. Tagad tiks izskatītas vairākas alternatīvas.

Atkarības starp testiem

Testpiemēri testu kopā var būt atkarīgi no citu testpiemēru izpildes rezultātā. To var specificēt katram testpiemēram piekārtojot norādi uz tiem testpiemēriem, kuriem obligāti ir jāizpildās pirms tā. Piemēram, ja mums ir četri testpiemēri, tās norādes varētu būt sekojošas:

$$\begin{gathered} t_1 : \{\} \\ t_2 : \{t_1\} \\ t_3 : \{t_1\} \\ t_4 : \{t_1, t_2\} \end{gathered} \tag{4.2}$$

Šādas atkarības varētu attēlot arī grafa veidā, kas ir parādīts 4.4. attēlā.



4.4. att. Testpiemēru atkarību grafs

Dotajā gadījumā bija lieki norādīt, ka t_4 ir atkarīgs no t_1, jo tas transitīvi seko no tā, ka t_4 ir atkarīgs no t_2, bet tas savukārt ir atkarīgs no t_1. Ņemot vērā tās atkarības, testu dziņa plānotājs varētu automātiski izveidot testpiemēru secību, kas atbilst šiem nosacījumiem. Dotajā piemērā tas varētu būt $D = (t_1 t_2 t_3 t_4)$, $D = (t_1 t_3 t_2 t_4)$ vai $D = (t_1 t_2 t_4 t_3)$. Gadījumā ar četriem testpiemēriem izsecināt iespējamās secības nav grūti. Ja testpiemēru ir ļoti daudz, tas varētu būt sarežģītāk, pie tam, mainoties testu kopas sastāvam, būtu jāmodificē secība, kas arī var nebūt triviāls uzdevums. Gadījumā ja testu dzinis pats prot uzbūvēt korektu secību, balstoties uz atkarībām, testa izņemšana vai pievienošana testu kopai reducējas uz to, ka ir jānorāda pareizas atkarības.

Atkarības ar nosacījumiem

Uz atkarībām balstītu modeli var paplašināt, piešķirot atkarībām papildus nosacījumus. Piemēram, varētu būt ne tikai atkarības no fakta, ka kāds testpiemērs jau ir izpildīts iepriekš, bet arī atkarības no tā testpiemēra izpildes rezultāta.

To var ilustrēt ar vienkāršu piemēru. Ja ir divi testpiemēri, pirmais no kuriem testē datu pievienošanu datubāzei, bet otrais — datu dzēšanu, tad saprātīgi būtu deklarēt otro testpiemēru par atkarīgu no pirmā (pirms dzēšanas dati ir jāpievieno). Taču šajā gadījumā ja pirmais testpiemērs nebūtu izgājis (datus neizdevās pievienot), visdrīzāk otrā testpiemēra pirmsnosacījumi nebūtu apmierināti. Tas dod pamatu veidot nevis atkarību no izpildīšanas fakta, bet atkarību no rezultāta.

Ja 4.4. attēlā parādītajā piemērā ceturtā testpiemēra atkarībai no pirmā piešķirt nosacījumu, ka ceturtais tests jāizpilda tikai, ja pirmais ir sekmīgi izgājis, tad šī atkarība vairs nebūtu lieka. Veidojas pavisam jauna situācija — testpiemēru secību testu kopā testu dzinis vairs nevar pateikt iepriekš, tas var tikai uzģenerēt izpildes plānu, kas tiktu modificēts testu kopas izpildes gaitā atkarībā no izpildīto testpiemēru rezultātiem. Piemēram, pēc izpildes varētu izrādīties, ka tika izpildīta secība $D = (t_1 t_3 t_2)$, un ceturtais tests vispār netika izpildīts, jo pirmais beidzās nesekmīgi.

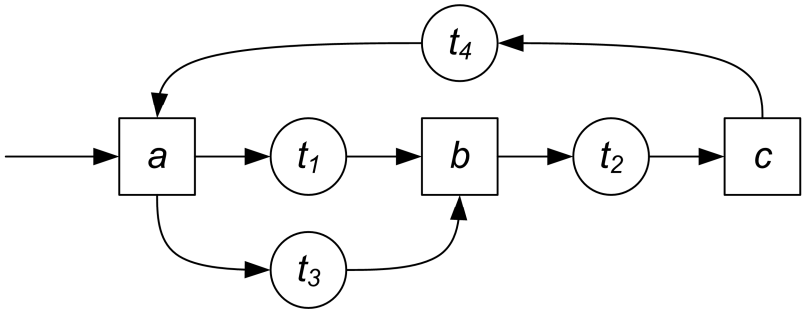
Stāvokļi

Mūsdienu interaktīvas programmas izpildoties var nonākt visdažādākos stāvokļos. Piemēram, programmās ar grafiskām lietotāja saskarnēm stāvokli varētu raksturoties ar dotajā brīdī atvērtajiem logiem, datu esamību vai neesamību tajos, apstrādājamo datu (dokumentu, datubāzu, utt.) stāvokli un citiem faktoriem [84]. Katru testpiemēru raksturo ne tikai tā dati (ievaddati un sagaidāmie rezultāti), bet arī stāvoklis, kurā testpiemēra izpildei jāsākas, un stāvoklis, kurā tai ir jābeidzas. Faktiski tāda gadījumā sākuma stāvoklis ir testpiemēra pirmsnosacījums, bet beigu stāvoklis — pēcnosacījums.

Šajā gadījumā būtu vērtīgi balstīt testpiemēru izpildes secības plānošanu uz informāciju par sistēmas stāvokļiem. Katram testpiemēram būtu papildus jādefinē sākuma un beigu stāvoklis. Tad testu dziņa plānošanas algoritms varētu izmantot šo informāciju, lai ģenerētu korektu testu kopas testpiemēru izpildes secību jeb testu komplektu. Pieņemsim, ka ir četri testi ar attiecīgiem sākuma un beigu stāvokļiem:

$$
\begin{aligned}
&t_1\colon (a, b)\\
&t_2\colon (b, c)\\
&t_3\colon (a, b)\\
&t_4\colon (c, a)
\end{aligned}
\tag{4.3}
$$

Pieņemsim arī, ka sākuma stāvoklis ir a. Tad šo informāciju par stāvokļiem varētu reprezentēt ar grafu, kā tas ir parādīts 4.5. attēlā.



4.5. att. Stāvokļu un testu grafs

Testu dziņa plānotājs varētu izveidot, piemēram, šādas izpildes secības: $D = (t_1 t_2 t_4 t_3)$ vai $D = (t_3 t_2 t_4 t_1)$. Ir viegli pamanīt to, ka šādas secības nav automātiski atkārtojamas, t.i., pēc to izpildes nevar tās izpildīt vēl vienu reizi. Tas ir tāpēc, ka abos variantos testu kopas izpilde beidzās stāvoklī b, t.i., sākuma stāvoklis nav nodrošināts.

Šo problēmu plānotājs varētu atrisināt atkārtoti izpildot otro un ceturto testpiemērus, piemēram, $D = (t_1 t_2 t_4 t_3 t_2 t_4)$. Šī secība lieki atkārto divus testpiemērus, taču nodrošina atkārtojamību, jo beidzās tajā pašā stāvoklī, kurā sākās (piemēram, tas stāvoklis varētu būt atvērtais programmas galvenais logs ar neaizpildītiem laukiem). Var viegli iedomāties gadījumus, kas vispār nebūtu izpildāmi bez šādas atkārtošanas, piemēram, pieliekot vēl vienu testu $t_5\colon (a, b)$, veseli trīs testpiemēri sāktos stāvoklī a. Tātad vismaz divas reizes būtu uz to jāatgriežas, bet to prot darīt tikai ceturtais testpiemērs. Tas nozīmē, ka ceturtajam testpiemēram būtu jāizpildās vismaz divreiz.

Liela testpiemēru skaita gadījumā testu dziņa plānotājam būtu papildus uzdevums — jāveido visīsākā (vai arī pieņemami īsa) secība, kas iziet cauri visiem testpiemēriem un, ja nepieciešams, atgriežas sākuma stāvoklī. Plānotājs var arī sastapties ar problēmu, ka secība vispār var neeksistēt. Ir divi gadījumi kad tā var notikt:

1) ja grafā ir stāvokļi, uz kuriem neeksistē ceļš no sākuma stāvokļa;
2) ja no kāda stāvokļa neeksistē ceļš uz sākuma stāvokli.

Otrais gadījums vienmēr kļūst par problēmu, kad pēc testu kopas izpildes ir jāatgriežas sākuma stāvoklī. Ja tāds nosacījums nepastāv, tad to var pārformulēt tā: grafā eksistē vismaz divi apakšgrafi bez kopīgiem stāvokļiem, tādi, ka neeksistē neviens testpiemērs, kas ved no vienam apakšgrafam piederošā stāvokļa uz otram apakšgrafam nepiederošo stāvokli. Vienkāršākajā gadījumā tas var izpausties tā, ka grafam ir vismaz divas virsotnes, no kurām nav izeju.

Šajos gadījumos plānotājs var atgriezt kļūdas paziņojumu ar attiecīgu informāciju. Tad testu kopai būtu jāpievieno papildus darbības (fiktīvus testpiemērus), kas nodrošina nepieciešamas pārejas. Plānotājs var arī izdot brīdinājumus gadījumos, kad testpiemēru secība varētu sanākt optimālāka, ja testu kopai būtu pievienota kāda papildus darbība, kas savieno divus noteiktus stāvokļus.

Stāvokļu sistēmas

Bieži ir grūti nomodelēt sistēmu ar vienu stāvokļu pāreju diagrammu. Piemēram, programmai, kurai ir grafiskā lietotāja saskarne un kura strādā ar datiem datubāzē, var būt salikti stāvokļi, piemēram, „atvērts galvenais logs, ieraksts datubāzē nepastāv“ vai „atvērts ieraksta modificēšanas logs, ieraksts datubāzē pastāv“.

Tādus gadījumus ir ērtāk apstrādāt, ja ieviest paralēlas stāvokļu sistēmas. Tad katram testpiemēram būtu jānorāda sākuma un beigu stāvokļi abās sistēmās. Testu dziņa plānotāja vienkāršāka realizācija šim gadījumam būtu tāda: sākumā apvienot abas stāvokļu sistēmas vienā, kur vienotās stāvokļu sistēmas stāvokļu kopa veidotos kā sākotnējo sistēmu stāvokļu kopu Dekarta reizinājums. Tad šis gadījums reducējas uz iepriekšējo.

4.3. Testu komplektu ģenerēšanas bibliotēka TSGL

Automātiskās testu komplektu ģenerēšanas jeb testu komplektu sastādīšanas no testpiemēriem, uzdevuma veikšanai tika izstrādāta bibliotēka TSGL (*Test Suite Generation Library*).

Atbilstoši 2.4. sadaļā aprakstītā vienotā automatizētās testēšanas modeļa apzīmējumiem TSGL nodrošina papildus funkcijas testu ģeneratoram un izpildes ietvaram. Tās izvade var būt testa gan testu XML formātā specificēta izpildes secība, kurai tādā gadījumā ir testa specifikācijas loma, gan tajā ir nodrošinātas funkcijas, kas spēj izsaukt testu izpildi šajā

secībā. Par ievadu kalpo nestrukturēta testu kopa un testu metadati, kuros ir ievietota informācija par katra testa sākuma un beigu stāvokli. Pašu testpiemēru izveidošana un izpildes nodrošināšana nav TSGL kompetencē, tam ir nepieciešami citi rīki, piemēram, kāds no darba 1. nodaļā minētajiem.

TSGL bibliotēkas pamatā ir dinamiskais testu dzinis, kas ir balstīts uz testējamās programmas stāvokļu informāciju. Dziņa galvenais komponents ir testu komplektētājs, kas no dotās testpiemēru kopas izveido testpiemēru secību tādā veidā, ka katrs nākamais tests sākās stāvoklī kurā beidzās iepriekšējais. Bibliotēkā ir iestrādāti divi komplektētāju veidi:

- balstīts uz vienkāršo stāvokļu sistēmām;
- balstīts uz salikto stāvokļu sistēmām.

Kaut arī uz salikto stāvokļu sistēmām balstītais komplektētājs spēj apstrādāt arī vienkāršo stāvokļu sistēmas kā speciālgadījumus, pirmais mehānisms vēsturiski tika izstrādāts pirmais, un tā darbība ir vienkāršāka, tāpēc ir saglabāti abi varianti. Šajā apakšnodaļā tiek aprakstītas koncepcijas, kas ir šo komplektētāju darbības pamatā.

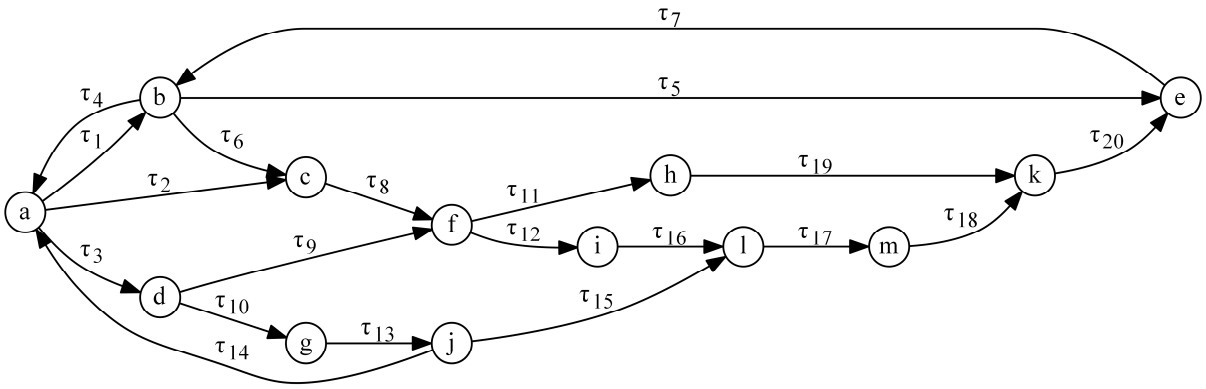
4.3.1. Testu komplektu ģenerēšana no vienkāršo stāvokļu sistēmas

Par ievadu testu komplektētājam kalpo testpiemēru kopa [119]. Taču no komplektētāja viedokļa būtisks ir nevis testpiemēru saturs, bet to metadati:

- testpiemēra sākuma stāvoklis, kurā ir jāsākas testpiemēra izpildei;
- testpiemēra beigu stāvoklis, kurā testpiemēram ir jābeidzas.

Komplektētāja darbība sastāv no sekojošajiem soļiem:

1) katram testpiemēram nolasa sākuma un beigu stāvokļu informāciju;
2) konstruē atmiņā stāvokļu pāreju grafu, kurā virsotnes atbilst testējamās programmas stāvokļiem, bet loki — testpiemēriem τ_i, kā 4.6. attēlā parādītajā piemērā;
3) konstruē ceļu grafā, kas sākās dotajā sistēmas sākumā stāvoklī, satur visus grafa lokus, un beidzās tajā pašā sistēmas sākuma stāvoklī — šis ceļš arī nosaka rezultāta testu komplektu, vai, precīzāk, loku secība ceļā nosaka testpiemēru secību komplektā.



4.6. att. Stāvokļu pāreju grafa piemērs

Nosacījums, ka ceļam ir jāsākas un jābeidzas vienā un tajā pašā stāvoklī, ir nepieciešams, lai nodrošinātu testu komplekta atkārtojamību. Ja komplekts beigtos citā stāvoklī, nebūtu iespējams izpildīt šo komplektu divas reizes pēc kārtas bez manuālās iejaukšanas.

Loki rezultāta ceļā drīkst parādīties vairāk, nekā vienu reizi. 4.6. attēlā dotajā piemērā aiz testa τ_{16} ir iespējama tikai testa τ_{17} izpilde, bet aiz testa τ_{15} — arī tikai testa τ_{17} izpilde. Tas nozīmē, ka testam τ_{17} būs jāparādās testu komplektā vismaz divas reizes, un tas ir pilnīgi pieņemami.

Kaut arī īsāki testu komplekti (tie, kas satur mazāku loku skaitu) ir labāki, nav nepieciešams izvēlēties visīsāko ceļu, jo ja testu komplekts izpildīsies automātiski, tas nepatērēs cilvēka laiku. Taču testu komplektam jebkurā gadījumā ir jābūt pieņemami īsam, jo komplekts, kura izpildei ir nepieciešamas trīs dienas būtu nepieņemams, ja eksistē divdesmit minūšu ekvivalents.

Ir viegli pierādīt, ka šāda komplekta eksistences nepieciešams un pietiekams nosacījums ir tāds, ka katram testpiemēram pastāv ceļš no dotā sistēmas sākuma stāvokļa līdz testpiemēra sākuma stāvoklim un pastāv arī ceļš no testpiemēra beigu stāvoklim līdz dotajam sistēmas sākuma stāvoklim. TSGL uz vienkāršajiem stāvokļiem balstītais komplektētājs izmanto šo faktu, meklējot un kombinējot šos ceļus, lai nodrošinātu katra loka iekļaušanas prasību.

4.3.2. Testu komplektu ģenerēšana no salikto stāvokļu sistēmas

Balstīta uz vienkāršajiem stāvokļiem testu komplektu ģenerēšanas metode ir ierobežota, it īpaši gadījumā ar lietotāja saskarnes līmeņa automatizētajiem testiem. Testam, kura mērķis

ir pārbaudīt, vai sistēmas administratoram ir iespēja mainīt kāda lietotāja paroli, varētu būt vairāki pirmsnosacījumi, piemēram:

1) lietotājam, kura parolei ir jābūt nomainītai, ir jābūt jau reģistrētam sistēmā;
2) aktīvajam lietotājam ir jābūt autentificētam kā sistēmas administratoram;
3) lietotāja profila dialoga logam ir jābūt atvērtam.

Tādus pirmsnosacījumus ir grūti modelēt ar vienkāršajiem stāvokļiem. Tā vietā ir izdevīgāk definēt šādus stāvokļus kā vārdu-vērtību pāru kopas [113]. Minētajā piemērā pirmsnosacījumi jeb sākuma stāvoklis varētu būt aprakstīts kā {„lietotājs A ir reģistrēts“ = true; „autentificētā loma“ = administrators; „aktīvais logs“ = „lietotāja A profils“}. Atsevišķus šādus vārdu-vērtību pārus sauksim par stāvokļa komponentiem. Vienkāršus stāvokļus tādā gadījumā var uzskatīt par saliktajiem stāvokļiem ar vienu stāvokļa komponentu, piemēram {„aktīvais stāvoklis“ = a}.

Var viegli pamanīt, ka stāvokļu komponenti, kas ir svarīgi vienam testam, var būt nesvarīgi citam. Piemēram, stāvokļa komponents „lietotājs A ir reģistrēts“ nav svarīgs testam, kas pārbauda, vai sistēmas administratoram ir iespēja konfigurēt e-pasta ziņojumu sūtīšanu. Ir iespējami trīs testa informētības tipi attiecībā uz stāvokļa komponenta vērtībām. Tie tipi tiks apzīmēti šādi:

- $R(x, y)$ — tests pieprasa noteiktu stāvokļa komponenta vērtību x kā priekšnosacījumu un uzstāda tam vērtību y kā pēcnosacījumu. Var būt arī, ka $x = y$, gadījumā, ja tests nemaina stāvokļa komponenta vērtību;
- $S(y)$ — testam nav svarīga stāvokļa komponenta vērtība (tas var strādāt ar jebkādu) kā priekšnosacījums, bet darbības rezultātā uzstāda tā vērtību par y kā pēcnosacījumu;
- U — testam nav svarīga stāvokļa komponenta vērtība. Jebkura vērtība apmierinās testa pirmsnosacījumus un tā paliek nemainīga kā pēcnosacījums.

Tiek pieņemts, ka visi testi ir determinēti, tāpēc jā noteikta stāvokļa komponenta vērtība ir definēta kā pirmsnosacījums, tad tā vai nu mainās ar citu noteiktu vērtību, vai nemainās, tātad tā nekad nepaliek nedefinēta.

Divus testus τ_1 un τ_2 ir iespējams savienot ceļa posmā $<\tau_1, \tau_2>$, ja nevienam stāvokļa komponentam nerodas konflikts starp τ_1 pēcnosacījumiem un τ_2 priekšnosacījumiem. Šādam ceļu savienojumam arī būs savi priekšnosacījumi un pēcnosacījumi un tāpēc arī informētības tips. Pēc līdzīga principa ceļa posmā var savienot ne tikai divus testus, bet arī testu ar jau esošo ceļa posmu vai divus ceļa posmus. Ja atsevišķus testus uzskatīt par ceļa posmiem, kas sastāv no viena testa, tad šī savienošanas operācija ir definēta uz ceļa posmu kopas.

4.1. tabula parāda kā no savienojamo ceļa posmu C_1 un C_2 informētības tipiem tiek veidots rezultāta ceļa posma $C_3 = <C_1, C_2>$ informētības tips.

4.1. tabula

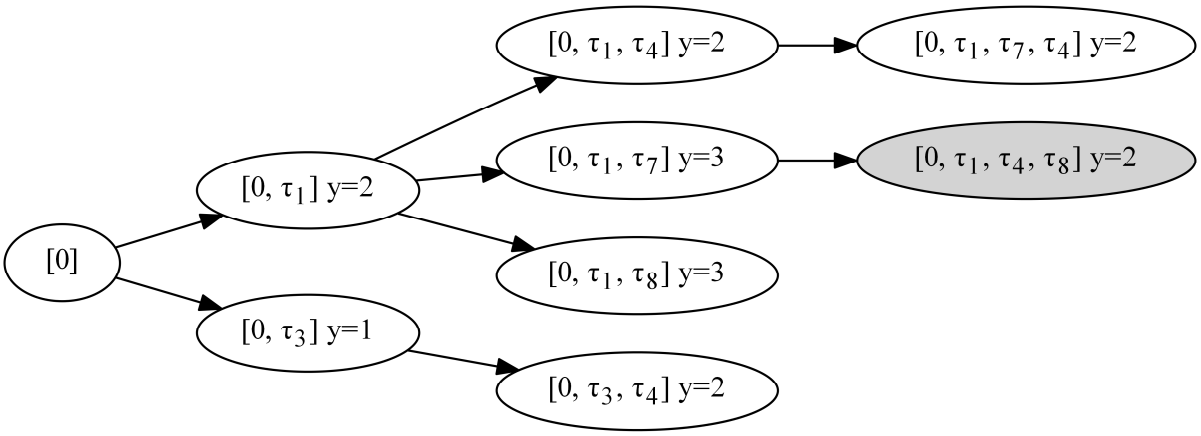
Ceļu posmu savienošanas likumi

Kreisā ceļa posma informētības tips	**Labā ceļa posma informētības tips**		
	R(x₂, y₂)	***S(y₂)***	***U***
R(x₁, y₁)	*R(x₁, y₂)*, ja $y_1=x_2$ savādāk – nav iespējams	*R(x₁, y₂)*	*R(x₁, y₁)*
S(y₁)	*R(x₂, y₂)*, ja $y_1=x_2$ savādāk – nav iespējams	*S(y₂)*	*S(y₁)*
U	*R(x₂, y₂)*	*S(y₂)*	*U*

Piemēram, ja tests τ_1 nepieprasa nekādu stāvokļa komponenta vērtību, bet uzstāda to, bet tests τ_2 arī nepieprasa vērtību bet uzstāda savu, savienojuma rezultāta informētības tipu var atrast pēc tabulas: $S(y_1)\ S(y_2) = S(y_2)$, tātad ceļa posmam $<\tau_1, \tau_2>$ pirmsnosacījuma nav, bet pēcnosacījums sakrīt ar τ_2 pēcnosacījumu.

Šādas savienošanas iespējamība nav transitīva. Ja ir iespējams izveidot ceļa posmu $<\tau_1, \tau_2>$ un ir iespējams izveidot ceļa posmu $<\tau_2, \tau_3>$, tas vēl nenozīmē, ka var izveidot ceļa posmu $<\tau_1, \tau_2, \tau_3>$. Piemēram, ja atbilstoši informētības tipi ir $S(y_1)$, U, $R(x_3, y_3)$, tad visus trīs testus nevar savienot ceļa posmā, ja $y_1 \neq x_3$. Savienošanas netransitivitāte nozīmē, ka uz grafiem balstītā pieeja nav derīga, jo testu savienošana ceļu posmos ir sarežģītāka, nekā ceļa izvēle grafā. TSGL realizē metodi, kas ģenerē testu komplektu, balstoties uz ceļu posmu koku.

1. Par koka sakni kalpo fiktīvas „nulles tests“. Nulles tests modelē sistēmas sākuma stāvokli, pasargājot sākuma vērtības visiem stāvokļa komponentiem. Nulles testam ir informētības tips $R(x, x)$ visiem stāvokļa komponentiem, jo ir nepieciešams, lai testu komplekts beigtos tajā pašā stāvoklī, kādā tas sākās. Tātad koka saknē ir ceļa posms, kas sastāv no viena nulles testa. Katrs koka mezgls reprezentēs kādu ceļa posmu.
2. Katrs nākamais koka līmenis tiek veidots no iepriekšējā. Katram līmeņa mezglam ir nepieciešams atrast tādus testus, kas var būt pielikti klāt (no labās puses) šim mezglam atbilstošajam ceļa posmam. Jauni ceļa posmi (jau par vienu elementu lielāki) kļūst par nākamā līmeņa mezgliem — dotā mezgla bērniem. Šo procesu ilustrē 4.7. attēls, kurā stāvoklis ietver vienu komponentu y ilustrācijas vienkāršības labā.



4.7. att. Ceļu koka fragmenta piemērs

Šis process ir jāatkārto kamēr netiks atrasts ceļa posms, kas satur visus testus un beidzās ar nulles testu. Algoritmam ir eksponenciālā sarežģītība un praksē, kad testu skaits ir liels, šī metode tās tīrā veidā nav piemērota. TSGL izmanto divas heiristikas, kas šo metodi padara par praktiskāku.

Pirmā heiristika ir pārlādes testa izmantošana. Praksē bieži pastāv universālā metode sistēmas stāvokļa pārlādēšanai sākotnējā. Ja tas ir iespējams, nulles tests var tikt realizēts kā pārlādēšana. Tāda pārlādes testa informētības tips būtu $S(x)$ katram stāvokļa komponentam. Tas nevar būt kandidāts pievienošanai esošiem ceļa posmiem. Ceļa augšanai ir jābeidzas brīdī, kad katrs tests tika iekļauts vismaz vienā ceļa posmā tekošajā līmenī. Tad tas kļūst relatīvi vienkārši izvēlēties minimālo ceļu posmu kopu un konkatenēt tos, iegūstot rezultāta testa komplektu. Pārlādes tests nodrošina, ka visi ceļa posmi sākās sistēmas sākuma stāvoklī.

Otrā heiristika ir pielietojama, kad pirmā jau tiek izmantota. Ja testu skaits ir liels, koks var ātri izaugt plašumā. Ja parādās divi vai vairāk ceļa posmi, kuros pēdējais tests ir viens un tas pats, un arī pēcnosacījumi šiem ceļa posmiem ir vienādi, tad tikai vienam no tiem (vislabāk, īsākajam) ir jāaug tālāk. Savādāk attiecīgo mezglu apakškoki būtu ekvivalenti no jauno testu sasniedzamības viedokļa. Piemērā (4.7. att.) iezīmētais ceļa posms tālāk neaugs. Tā pēdējais tests ir τ_4, bet pēcnosacījumi ir $\{y=2\}$ — kokā jau ir mezgls ar tādu pašu pēdējo testu un tādiem pašiem pēcnosacījumiem.

4.4. Ceturtās nodaļas secinājumi

4.4.1. Piedāvāts klasificēt testu dziņus statiskajos (testpiemēru izpildes secība ir iepriekš viennozīmīgi definēta) un dinamiskajos (testpiemēru izpildes secība tiek ģenerēta to izpildes gaitā). Analizēti katras klases dziņu veidi un paveidi, kā arī izstrādāti to modeļi.

4.4.2. Izstrādātas divas testu komplektu ģenerēšanas metodes — balstītas uz vienkāršo stāvokļu sistēmām un uz salikto stāvokļu sistēmām. Pirmā metode ir lietojama, ja testpiemēriem ir iespējams norādīt konkrētus vienkāršus sākuma un beigu stāvokļus, tādā veidā radot stāvokļu pāreju sistēmu. Otrā metode ir pirmās paplašinājums un ir derīga gadījumiem, kad testpiemēru sākuma un beigu stāvokļi ir reprezentējami kā vairāku stāvokļu kombinācijas no dažādām ortogonālām stāvokļu pāreju sistēmām.

4.4.3. Izstrādāta TSGL bibliotēka, kurā ir realizētas abas piedāvātās metodes. Bibliotēku ir iespējams izmantot automātiskajai testu komplektu ģenerēšanai no testu kopas gadījumos, kad atkarības starp testiem ir reprezentējamas kā vienkāršo vai salikto stāvokļu sistēmas.

5. VEIKTSPĒJAS TESTĒŠANAS RĪKA PICUS IZSTRĀDE

5.1. Pielāgojamā veiktspējas testēšanas rīka izstrāde un paplašināšana

5.1.1. Veiktspējas testēšanas uzdevums un rīku īpašības

Programmatūras veiktspējas testēšana ir svarīga vairāklietotāju sistēmu kvalitātes novērtēšanas sastāvdaļa. Lai notestētu sistēmas veiktspēju ir nepieciešams savākt veiktspējas metrikas vairāku lietotāju vienlaicīga darba apstākļos. Tādā veidā ir iespējams iegūt informāciju par dažādu slodzes līmeņu ietekmi uz atsevišķu lietotāju novēroto sistēmas uzvedību.

Veiktspējas testu projektēšana ievērojami atšķiras no funkcionālo testu projektēšanas [30]. Testpiemēru funkcionālās detaļas, tādas kā ievaddati, nav tik svarīgas veiktspējas testēšanā. Daudz svarīgāki aspekti ir lietotāju darba scenāriji un ģenerējama slodze. Mērķa rezultāti arī ir atšķirīgi. Veiktspējas testi nepievērš daudz uzmanības sistēmas darbību funkcionālajam korektumam, tie vairāk ir vērsti uz atbildes laikiem, procesoru un atmiņas izmantošanu, pieprasījumu rindu garumiem un citiem ar veiktspēju saistītiem rādītājiem. Mūsdienās uzmanība tiek vairāk pārorientēta no veiktspējas metrikām, kuras ir iegūstamas sistēmas līmenī, uz tādām, kas ir vērojamas gala lietotāja līmenī [94]. Par lapas kļūmju intensitāti datubāzes serverī vai tīkla caurlaidību gala lietotājs nezina un nerūpējas. Kas ir tiešām viņam svarīgs — tas ir vērojams sistēmas atbildes laiks un vispārējs atbilžu korektums.

Veiktspējas testēšana ir cieši saistīta ar drošumu un mērogojamību, un adekvāta veiktspējas testēšana ir spējīga palīdzēt uzlabot šos un arī citus kvalitātes aspektus [10]. Kaut arī pastāv metodes, kas palīdz novērtēt veiktspēju pat pirms sistēma tika izstrādāta [127], tomēr objektīvākus mērījumus ir iespējams iegūt, tikai novērtējot sistēmu tās reālas ekspluatācijas vidē. Pastāv ļoti daudz faktoru, kas ietekmē veiktspēju, tādi kā tīkls, servera un klienta vides [110], bet to ietekmi pilnībā paredzēt pirms sistēmas izvietošanas (*deployment*) nav iespējams. Veiktspējas testēšanas rīki ir paredzēti sistēmas veiktspējas novērtēšanai reālas ekspluatācijas vidē vai testēšanas vidē, kas ir maksimāli pietuvināta reālajai.

Veiktspējas testēšana no gala lietotāja viedokļa nozīmē pirmkārt to, ka testējama sistēma tiek noslogota ar darbībām, kas atbilst reālā lietotāja ģenerētajām, un otrkārt to, ka

tiek mērīti veiktspējas rādītāji, kuru izpausmi lietotājs tiešā veidā izjūt — sistēmas reakcijas ātrums un korektums.

Tātad divas pamatfunkcijas, kas jānodrošina šajā rīku klasē, ir:

- reālā lietotāja darbību emulēšana, kā arī spēja emulēt vairāku lietotāju darbu vienlaicīgi;
- sistēmas atbilžu laika un tā korektuma mērīšana.

Veiktspējas testa veikšanai ir nepieciešams skripts, kas nosaka, kādas darbības veic emulētais lietotājs, ko un kad tas pārbauda iegūstamajās atbildēs. Veiktspējas testa skripts strādā protokola līmenī, t.i., emulē komunikāciju, kas notiek starp klienta un servera programmatūru, lai izvairītos no nepieciešamības darbināt klienta programmatūru testa laikā, tādējādi taupot sistēmas resursus.

Tā kā tās divas pamatfunkcijas arī nosaka rīka piederību šai rīku klasei, tās pēc definīcijas nodrošina visi šādi rīki. Dažādi šīs klases rīki atšķiras gan pēc licenču cenām, gan pēc vairāku papildus funkciju nodrošināšanas, kas var būt būtiskas konkrētiem projektiem. Starp šādām papildus funkcijām var minēt sekojošas.

- Skripta ierakstīšanas iespēja. Automatizē skripta sagataves izveidi, pārķerot un ierakstot komunikāciju starp klientu un serveri.
- Protokolu atbalsts. Visvairāk rīki atbalsta tīmekļa protokolus. Taču ir rīki, kas nodrošina darbu ar vairākiem dažādiem protokoliem.
- Komunikācijas dinamiskuma atbalsts. Vienādām lietotāja darbībām ne vienmēr atbilst vienāda komunikācija ar serveri. Dažas vērtības katru reizi var manīties, spilgtākais piemērs — sesijas numurs. Rīku spējas apstrādāt šādas vērtības ir atšķirīgas.
- Korektuma pārbaužu veidi. Dažādi rīki piedāvā dažādas iespējas pārbaudīt, vai no servera iegūtas atbildes ir korektas. Piemēram, HTTP atbilžu kodi, noteikta teksta parādīšanās atbilde, noteikta atbildes struktūra utt.
- Slodzes scenāriju uzbūves veidi. Ir rīki, kas pieļauj tikai vienu skriptu scenārijā, citi — ne tikai atļauj vairākus skriptus, bet arī ļauj katram skriptam uzdot savu laika plānu.
- Attālas pārraudzības atbalsts. Ir rīki, kas spēj attāli vākt mērījumus no sistēmas serveriem, piemēram, operētājsistēmas veiktspējas rādītājus, tīmekļa un datubāzes rādītājus utt.
- Atbalstītas platformas. Daži rīki ir piesaistīti konkrētai platformai (piemēram, Windows), citus var darbināt vairākās platformās.
- Slodzes sadalīšanas iespējas. Ja ir nepieciešama ļoti lielas noslodzes ģenerēšana, vienas darbstacijas resursu var nepietikt. Daži rīki ļauj uzstādīt vairākus slodzes ģeneratorus

(slodzes aģentus) dažādās darbstacijās un vienā testā ģenerēt noslodzi no vairākiem punktiem.

- Un vairākas citas.

Veiktspējas testēšanas rīka funkcionalitāte var kalpot dažādiem mērķiem. Veiktspējas testēšanai kā tādai ir vairāki paveidi, kuriem ir dažādi mērķi [109]. Piemēram, veiktspējas testēšanas paveidi ir šādi:

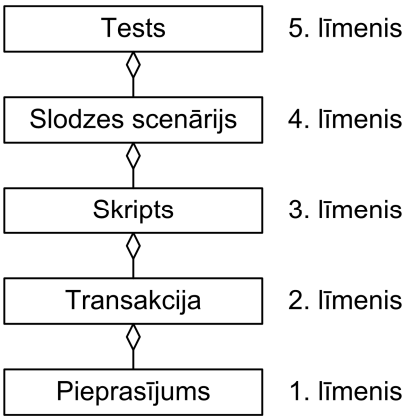
- kapacitātes tests — tests, kura mērķis ir noteikt slodzes līmeni, pie kura pārtrūkst kāds svarīgs sistēmas resurss (atmiņa, tīkla caurlaidspēja utt.);
- izturības tests — ilgstošs tests, kura mērķis ir noteikt, kā veiktspējas īpašības mainās ar laiku, pie reālas darbināšanas normālas slodzes;
- lēcienu tests — tests, kas fokusējas uz veiktspējas īpašību noteikšanu scenārijos, kad slodze pēkšņi uz īsu brīdi kļūst lielāka par slodzi reālas darbināšanas apstākļos;
- stresa tests — tests, kura mērķis ir noteikt testējamās sistēmas darbspēju apstākļos kad slodze kļūst lielāka par normālo un/vai maksimālo;
- un citi.

Tātad slodzes testēšanas paveidi atšķiras ar tiem mērķiem, kas ir jāsasniedz. Testu paveidi nosaka slodzes scenārija izvēli. Piemēram, izturības testam ir jābūt pietiekoši ilgam, savukārt stresa tests var būt samērā īss, taču ģenerējamai slodzei jābūt daudz lielākai. No mērķiem dažreiz var būt atkarīga arī skriptu izvēle. Piemēram, ja ir nepieciešams veikt stresa testu tikai kādam noteiktam sistēmas slānim (datubāzes serveris, tīmekļa serveris utt.), skripti ir jāizvēlas tādi, kas vairāk noslogos tieši šo slāni.

Slodzes testēšanas rīka piemērotība kādam konkrētam testēšanas paveidam ir atkarīga no rīka iespējām — pietiekoši elastīgam rīkam ir iespēja veikt visus minētus testu paveidus.

5.1.2. Veiktspējas testa modelis

Veiktspējas testa konkrēta struktūra ir atkarīga no lietojamiem rīkiem. Šeit tiks aprakstīts konceptuāls, no rīkiem neatkarīgs veiktspējas testa modelis, uz kura tiks balstīti tālākie apraksti. Modelis, kas ietver piecus līmeņus, ir parādīts 5.1. attēlā.



5.1. att. Veiktspējas testa modelis

Programmatūra, kas veic lietotāju emulāciju testa laikā, kā arī aparatūra, uz kuras tā programmatūra darbojas, turpmāk tiks saukta par „slodzes aģentu".

1. līmenis — pieprasījumi

Zemākajā līmenī tests sastāv no pieprasījumiem. Veiktspējas testi parasti emulē lietotājus protokola līmenī, lai būtu iespēja darbināt virtuālos lietotājus bez klienta programmatūras iesaistīšanas. Šāda pieeja dod iespēju efektīvāk izmantot slodzes aģenta mašīnas resursus un darbināt daudz vairāk virtuālos lietotājus, nekā tas būtu iespējams, ja katram virtuālajam lietotājam būtu jādarbina savs klienta programmatūras eksemplārs. Pieprasījums ir dati, ko slodzes aģents sūta testējamajai sistēmai caur tīklu un kuriem testējamā sistēma veido un sūta atpakaļ atbildi. Tīmekļa gadījumā pieprasījumi būtu HTTP protokola GET un POST pieprasījumi.

Pieprasījumu līmenī testi var saturēt papildus soļus, tādus kā atbilžu korektuma verifikācija pēc noteiktiem likumiem, dinamisko komunikācijas elementu, piemēram, sesijas identifikatoru, apstrāde, un citus.

2. līmenis — transakcijas

Nākamais līmenis ir transakcijas. Transakciju var definēt kā atsevišķu operāciju, kuru veic testējamā sistēma un kas ir nedalāma no gala lietotāja viedokļa. Piemēram, pieteikšanās transakcija ir tas, kas notiek starp momentu, kad lietotājs nospiež pogu „Pieteikties", un momentu, kad tīmekļa portāla pirmā autorizēta lapa tiek pilnībā attēlota lietotāja

pārlūkprogrammā. Kaut arī no servera viedokļa transakcija ir vienkārši pieprasījumu virkne, kas ir jāapstrādā, no lietotāja viedokļa tā izskatās kā viena saturīga darbība.

Šī iemesla dēļ, testā, kas ir orientēts uz lietotājiem, transakciju izpildes laiki ir svarīgāki par atsevišķu pieprasījumu apstrādes laikiem.

3. līmenis — skripti

Nākamajā līmenī transakcijas tiek kombinētas skriptos. Skripts ir transakciju virkne, kas tiek izpildītas fiksētā vai mainīgā secībā.

Skripta nolūks ir emulēt tādas darbību virknes, ko var veikt reālie lietotāji, lai padarītu testu par reālistiskāku. Skriptā parasti tiek specificētas fiksētas vai mainīgas aiztures (zināmas arī kā domāšanas laiki) starp transakcijām ar mērķi reālistiskāk emulēt lietotāja aktivitātes — attēloto datu apskatīšanu, formu aizpildīšanu vai vienkārši peles kustināšanu un klikšķināšanu.

Tests var saturēt vairākus skriptus, katrs no kura emulē savu lietotāja veidu. Tātad virtuālais lietotājs ir dinamiski darbināma noteikta skripta instance.

4. līmenis — slodzes scenāriji

Slodzes scenārijs ir specifikācija tam, cik ilgi tests izpildīsies un kā virtuālo lietotāju skaits mainīsies laikā. Atkarībā no testa mērķa slodzes scenārijs varētu definēt ilglaicīgu fiksēta lietotāju skaita darbināšanu vai pakāpenisku lietotāju skaita palielināšanu no nulles līdz noteiktajam maksimumam. Šo divu piemēru variācijas, kā arī sarežģītāki slodzes scenāriji arī ir iespējami.

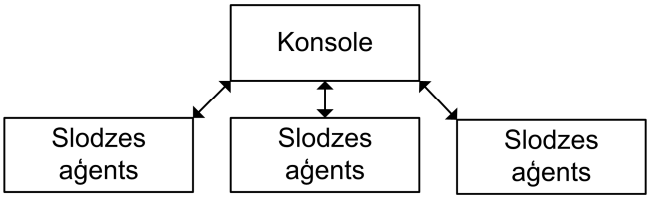
Augstākajā 5. līmenī tests var sastāvēt no vairākiem slodzes scenārijiem, katrs paredzēts izpildīšanai savā slodzes aģenta mašīnā, lai varētu emulēt lielas noslodzes, kurām nepietiktu ar vienas mašīnas resursiem.

5.1.3. Picus rīka arhitektūra un realizācija

Darba gaitā tika izstrādāts veiktspējas testēšanas rīks Picus, balstoties uz autora vairāku gadu pieredzi veiktspējas testēšanas pakalpojumu sniegšanā, izmantojot gan brīvi pieejamus, gan komerciālus rīkus, un ņemot vērā to stiprās un vājās puses [111]. Daudzi brīvi pieejami

atvērtā koda rīki nodrošina paplašināšanas iespējas, taču dažos projektos tās izrādījās nepietiekamas. Tāpēc tika pieņemts lēmums izstrādāt savu rīku, liekot īpašu uzsvaru uz tā paplašināmību, un attīstīt to pēc nepieciešamības, ko nosaka aktuālie projekti.

Lai būtu iespējams nodrošināt smagu slodzi testējamajai sistēmai, Picus rīkā tika ieviesti divi slāņi: konsole un slodzes aģenti. Caur konsoli ir iespējams konfigurēt, palaist un kontrolēt testus. Slodzes aģenti noslogo testējamo sistēmu un vāc veiktspējas statistiku, tādu kā atbilžu laiki un korektums. Slodzes aģenti var atrasties vairākās mašīnās, un to darbību kontrolē konsole. Rīka struktūra ir parādīta 5.2. attēlā. Gan konsole, gan slodzes aģenti ir implementēti Java valodā.



5.2. att. Rīka Picus struktūra

Konsole komunicē ar slodzes aģentiem caur TCP protokolu, ieskaitot komandu sūtīšanu un statistikas saņemšanu. Ir arī iespējams izpildīt slodzes aģentu konsoles procesā, tādā gadījumā tie komunicē taisni caur metožu izsaukumiem.

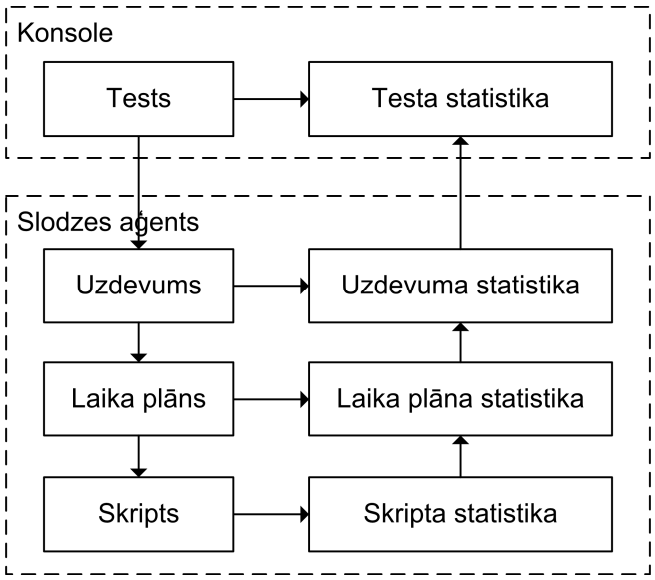
Lai varētu testēt konkrētu sistēmu, ir jādefinē testi. Testi Picus rīkā tiek definēti četros slāņos.

1. Skripts. Picus rīkā skripts ir Java klase, kas nosaka soļu secību, kuros tiek veiktas darbības ar testējamo sistēmu, vai tiek pārbaudīts saņemto atbilžu korektums. Skripts ir paredzēts reālā lietotāja emulēšanai, kurš veic noteiktas darbības ar sistēmu.
2. Laika plāns (*schedule*). Specificē, kā noteikta skripta instancēm ir jāizpildās: cik skripta instances ir jāizpilda (emulējamo lietotāju skaits), kā šis skaits mainās laikā (palielinās vai samazinās, palielinās un pēc tam samazinās, utt.), cik ilgi šādas izmaiņas turpinās un citus aspektus.
3. Uzdevums. Specificē noteiktam slodzes aģentam, kādus laika plānus tam jāizpilda.
4. Tests. Specificē, kā uzdevumi ir sadalīti starp pieejamajiem slodzes aģentiem.

Kaut arī pats skripts ir Java klase, cita testa informācija (laika plāni, uzdevumi un pats tests) tiek glabāti XML failā, un kura saturu var kontrolēt no konsoles lietotāja saskarnes.

Testa izpildes laikā katrs slānis vāc savu statistiku un sūta to attiecīgajam statistikas apstrādātāja komponentam. Katram slānim ir savs statistikas apstrādātājs. Zemākā līmeņa

komponents sūta statistiku austākā līmeņa komponentam, kurš to apstrādā, apstrādātus datus kombinē ar sava līmeņa statistiku un tad sūta nākamā augstāka līmeņa statistikas apstrādātājam. Šī datu plūsma ir ilustrēta 5.3. attēlā. Katra slāņa statistikas apstrādātājus var kontrolēt no konsoles: datu detalizācijas līmeni, kas jāsūta augstākajam līmenim, vai šī statistika ir jāraksta žurnālfailā, vai parādīt to konsoles lietotāja saskarnē utt.



5.3. att. Datu plūsmas starp slāņiem

Picus nodrošina paplašināšanas iespējas vairākos aspektos. Pašreiz ir skaidri definētas lietojumprogrammu saskarnes (API) priekš:

1) protokoliem. Skripti izmanto protokolus, lai ģenerētu pieprasījumus testējamajai sistēmai un lai analizētu tās atbildes. Jaunu protokolu implementācijas var pielikt Picus spraudņu veidā;
2) laika plāniem. Dažādi laika plānu tipi ir iespējami. Sākotnēji tika implementēti laika plāni, kas var izpildīt vienu skripta instanci vienu reizi, izpildīt N skripta instances fiksētā laika periodā, pakāpeniski palielināt aktīvo skripta instanču skaitu no nulles līdz N fiksēta laika periodā. Citi laika plānu tipi var tikt izstrādāti un pielikti Picus spraudņu veidā;
3) statistikas apstrādātājiem. Statistikas apstrādātāji nosaka stratēģiju, kā veiktspējas statistikas dati tiek apstrādāti un nodoti augstāka slāņa apstrādātājiem. Pašreiz ir realizēts viens statistikas apstrādātājs katram slānim, un pagaidām ar to bija pieticis visos izmantošanas gadījumos. Taču ir iespēja izstrādāt papildus apstrādātājus un pievienot tos Picus spraudņu veidā;

4) komunikatoriem. Komunikators nosaka, kādā veidā konsole sadarbojas ar slodzes aģentiem. Pašreiz ir izstrādāti divu komunikatoru veidi: komunikācijai caur TCP protokolu un komunikācijai tiešo metožu izsaukumu veidā.

Pēc 1.3. sadaļā izstrādātās klasifikācijas Picus pieder klasēm D3 (atbalsta datu tabulas), S1a-Java (izmanto Java programmēšanas valodu skriptu izveidei), M2 (atbalsta paralēlu skriptu izpildi), I3-HTTP (strādā protokola līmenī, pagaidām atbalstot tikai HTTP). Atbilstoši 2.4. sadaļā izstrādātā vienotā automatizētās testēšanas modeļa apzīmējumiem Picus ir tipiska izpildes ietvara realizācija, kas var būt pielāgojama dažāda veida testu specifikāciju apstrādei.

5.1.4. Picus izmantošana

Šajā sadaļā tiek aprakstīts veiktspējas testēšanas process pa posmiem, kā tas ir veicams ar Picus rīka palīdzību. Dažu posmu izpildei nepietiek ar Picus funkcionalitāti, tāpēc tajos varētu būt nepieciešama citu palīgrīku izmantošana.

Prasību noskaidrošana

Pats pirmais posms ir testa mērķu noskaidrošana. Tie var būt dažādi — pārliecināties par veiktspējas prasību apmierināšanu, noskaidrot šauras vietas sistēmā, identificēt funkcionālas problēmas, kas var rasties augstas slodzes apstākļos utt.

Atkarībā no mērķu formulējuma tiek izvēlētas tās lietotāju darbības, kas tiks realizētas skriptos. Rezultātā rodas skriptu apraksti. Katram izvēlētajam skriptam tiek precīzi formulētas tas darbības, ko veic lietotājs ar sistēmu, piemēram, kādas sistēmas datu meklēšanas skripta apraksts varētu sastāvēt no šādiem soļiem: pieteikšanās sistēmai, meklēšanas formas atvēršana, meklēšanas kritēriju aizpildīšana un meklēšanas iniciēšana, meklēšanas rezultātu apskatīšana, izeja no sistēmas. Katram solim tiek precīzi norādīts, kādas darbības jāveic — kādas pogas vai saites jāspiež, kādi formu lauki jāaizpilda ar kādām vērtībām utt.

Tiek arī noskaidrots, kā skripts tiek sadalīts sākuma, pamata un beigu daļās — t.i., kura aprakstītā skripta daļa izpildīsies iteratīvi testa laikā. Minētajā piemērā varētu būt tā, ka pieteikšanās sistēmai tiek ievietota sākuma daļā, izeja no sistēmas — beigu daļā, viss pārējais pamata daļā — tātad virtuālie lietotāji pieteiksies sistēmai, pēc tam veiks meklēšanu visa testa garumā un tikai pašās testa beigās izies no sistēmas. Cits variants, lai slodzes testā būtu lielāks

pieteikšanas un izejas darbību īpatsvars, visas darbības var ievietot pamata daļā, t.i., virtuālie lietotāji pieteiksies sistēmai un izies no tās katrā iterācijā.

Komunikācijas ierakstīšana

Kad skriptu saturs un soļi ir noskaidroti, ir nepieciešams veikt komunikācijas analīzi. Tā kā slodzes testa skripti ir veidojami protokola līmenī, aprakstītas darbības tagad ir jāformulē HTTP komandu terminos.

Picus rīkam pašlaik nav savas komunikācijas ierakstīšanas funkcionalitātes, tāpēc šim mērķim jāizmanto kāds papildrīks. Tas varētu būt WebLoad, kuram ir ierakstīšanas funkcionalitāte, kas pieņemami strādā vairākumā gadījumu. Cits labs rīks šim mērķim ir komerciāls rīks HttpAnalyzer, kas speciāli paredzēts HTTP komunikācijas analīzei.

Šis process notiek tā, ka tiek palaista attiecīga rīka komunikācijas pārķeršanas funkcija un tiek izpildītas darbības, kuras ir formulētas skripta prasībās. Kad tas ir izdarīts, komunikācijas analīzes rīks parāda, kādi pieprasījumi (HTTP komandas) tika sūtīti testējamas sistēmas serverim un kādas atbildes sistēma sniedza.

Skripta sākotnēja veidošana

Skripta veidošana Picus rīkā notiek tā, ka tiek radīta jauna AbstractScript klases apakšklase ar trim funkcijām — setup(), body() un teardown(), kuras paredzētas attiecīgi skripta sākuma, pamata un beigu daļas realizācijai. Tad izmantojot Picus skriptu veidošanas lietojumprogrammu saskarni, tiek rakstīts skripts, ņemot vērā iepriekšējā solī ierakstīto komunikācijas informāciju.

Šo procesu var saukt par komunikācijas „tulkošanu“ Picus saprotamajā valodā, ko var daļēji automatizēt. Rezultātā tiek iegūta Java klase, kas precīzi atspoguļo to komandu secību, kura tika noskaidrota komunikācijas ierakstīšanas posmā. Šeit tā komandu virkne tiek arī sadalīta pa transakcijām, ņemot vērā, kādas komandu apakšvirknes no gala lietotāja viedokļa izskatīsies kā viena darbība. Tātad izveidoto transakciju kopa praktiski vienmēr atbilst tai darbību kopai, kas tika definēta skripta aprakstā prasību noskaidrošanas posmā.

Skripta pilnveidošana

Iepriekšējā posmā izveidotais skripts gandrīz nekad nestrādā pareizi uzreiz, jo praktiski jebkurā skriptā parādās dinamiskas vērtības, kas mainās katrā skripta izpildes reizē. Tipisks piemērs ir sesijas numurs — kad lietotās pieteicās sistēmai, tā izveido sesiju un atgriež lietotājam (pārlūkprogrammai) numuru, kas unikāli identificē šo sesiju. Izpildot nākamās komandas, pārlūkprogramma var sūtīt šo numuru kā komandu argumentu. Tāpēc mēģinot izmantot iepriekšējā posmā veidoto skriptu (kā precīzu ierakstītas komunikācijas atkārtošanu), lietojot to sesiju numuru, kas tika izmantots komunikācijas ierakstīšanas brīdī, sistēma uzvedīsies nepareizi un atgriezīs kļūdainas atbildes, kas brīdinās, ka šī sesija jau ir beigusies. Lai tā nebūtu, skriptam ir jāprot analizēt no sistēmas iegūtas atbildes, atrast tajās jaunu sesijas numuru un izmantot to veca numura vietā.

Sesijas numurs ir tipisks, bet ne vienīgais dinamisko vērtību piemērs. Dinamisko vērtību apstrāde nav triviāls process, bieži tā nodrošināšanai ir jāpaļaujas uz intuīciju. Taču ir standartpaņēmiens, ko var izmantot vairumā gadījumu un kas gandrīz vienmēr palīdz korekti apstrādāt visas dinamiskas vērtības skriptā.

Šo paņēmienu var formulēt, aprakstot tā soļus.

1. Atkārtoti jāieraksta komunikācija tai pašai darbību virknei.
2. Jāsalīdzina izsūtāmo komandu saturs sākotnēji ierakstītai un atkārtoti ierakstītai komunikācijai.
3. Ja ir konstatēts, ka pirmajā un otrajā reizē kādas komandas saturs (ieskaitot argumentus, un POST komandas ķermeni) atšķiras, tad tā atšķirīga daļa ir dinamiska vērtība, kura ir speciāli jāapstrādā skriptā.
4. Atšķirīgo daļu meklējam kādas iepriekšējas komandas atbildē. Tā vērtība nevarēja parādīties izejošajā komandā pati par sevi — pārlūkprogramma visdrīzāk šo vērtību saņēma no sistēmas kādā no iepriekšējām atbildēm.
5. Kad tiek atrasta tā komanda, kuras atbilde satur šo vērtību, pēc tas komandas izpildes funkcijas skriptā ieliekam extractParameter() funkcijas izsaukumu no Picus API, kas šo vērtību nolasīs no atbildes un ievietos parametru vārdnīcā. Tad tai komandai, kurā šī vērtība tiek izmantota, aizvietojam konstantu vērtību ar Picus API funkcijas parameter() izsaukumu, kas ievietos vajadzīgajā vietā vajadzīgo mainīgo vērtību.
6. Process ir jāturpina no 2. soļa, kamēr nav apstrādātas visas komandas, kuru saturā ir konstatējamas atšķirības.

Šī metode dod labus rezultātus, taču tā negarantē visu dinamisku vērtību atrašanu. Piemēram, ja kāda vērtība komunikācijā apzīmē šīsdienas datumu, tad atšķirības varēs konstatēt tikai ja komunikācijas ierakstīšana notika dažādās dienās.

Kad visas dinamiskas vērtības ir apstrādātas, skriptu jau var izpildīt. Taču pirms tam ir vērts veikt pēdējo skripta pilnveidošanas soli — atbilžu korektuma pārbaudi. Šim mērķim Picus API ir funkcija checkText(), kas pārbauda, vai sistēmas atbildē uz noteiktu komandu ir atrodams noteikts teksts. Šādas pārbaudes ir lietderīgi paredzēt visām tām atbildēm, kurām ir dinamisks HTML saturs.

Skripta validācija

Skripta validācija ir process, kurā tiek konstatēts, vai skripts ir izstrādāts pareizi un atbilst pret to izvirzītām prasībām. Skriptu var uzskatīt par korektu, ja tas iziet trīs šādus validācijas soļus.

1. Skripta vienreizēja izpilde. Skripts tiek izpildīts vienu reizi pēc shēmas setup() → body() → teardown().
2. Skripta izpilde vairākās iterācijās. Skripts tiek izpildīts, izturot vairākas iterācijas (pietiek ar trim), piemēram, pēc shēmas setup() → body() → body() → body() → teardown(). Ar to tiek pārbaudīts, vai skripts spēj izpildīt body() daļu vairākas reizes pēc kārtas, t.i., vai iepriekšējā izpilde neietekmē nākamās iterācijas.
3. Skripta izpilde vairākos eksemplāros. Tiek izpildīts scenārijs, kurā strādā vairāki virtuālie lietotāji skaitā 5 līdz 10. Ar to tiek pārbaudīts, vai patiešām skripts ir izstrādāts tā, ka vairāki no tā veidotie virtuālie lietotāji spēj darboties vienlaicīgi (vai nav sinhronizācijas problēmu, vai viens virtuāls lietotājs neietekmē cita darbspēju utt.)

Ja validācijas posmā tiek konstatētas problēmas, tas nozīmē, ka skripts ir izstrādāts nekorekti un to ir nepieciešams papildus modificēt. Lai varētu analizēt problēmu cēloņus, ir paredzēts mehānisms, ieslēdzot kuru, tiek saglabātas pilnīgi visu komandu un to atbilžu galvenes un ķermeņi. Skatoties tos, un salīdzinot tos ar komunikācijas ierakstīšanas posmā iegūtiem datiem, var sameklēt atšķirības un secināt, kādi uzlabojumi skriptā ir nepieciešami.

Kad šis posms ir izpildīts, skripts ir novalidēts un gatavs lietošanai scenārijos. Tādu skriptu var izmantot vairākos scenārijos, atkārtot slodzes testus utt. Skripta modificēšana var būt nepieciešama tikai tad, kad mainās testējamas sistēmas versija. Šādā gadījumā komunikācija var mainīties un skriptu būs nepieciešams adaptēt, lai tas atbilstu jaunajam komunikācijas variantam.

Testa konfigurācijas izveide

Kad visi nepieciešami skripti ir izveidoti, var izveidot testa konfigurāciju. Tas posms ir samērā vienkāršs. Katram skriptam tiek izveidots atsevišķs laika plāns atbilstoši veiktspējas testa prasībām, pēc tam tie laika plāni tiek iekļauti uzdevumos un tie savukārt — testā.

Pārraudzības konfigurēšana

Picus nenodrošina nekādas sistēmu pārraudzības funkcionalitātes. Taču tāda pārraudzība ir būtiska testējamas sistēmas veiktspējas analīzei.

Būtiskākie rādītāji, ko ir vērts analizēt jebkurai testējamai sistēmai ir procesora noslodze, procesu rindas garums, operatīvās atmiņas patēriņš, atmiņas pārnešanas (*swapping*) biežums, tīkla noslodze. Ja zina, kā slodze ietekmē šos rādītājus, var viegli secināt, kādas pastāv šauras vietas sistēmā. Ir daudz citu rādītāju, kuras ir vērts analizēt — gan tie, ko sniedz operētājsistēma, gan tie, ko sniedz sistēmai specifiska programmatūra. Piemēram, datubāzes serveris var sniegt informāciju par aktīvo sesiju skaitu, SQL vaicājumu skaitu sekundē, lasīšanas un rakstīšanas operāciju skaitu sekundē utt.

Tā kā Picus pats šos rādītājus nevāc, ir nepieciešams lietot kādus citus rīkus, kas slodzes testa laikā varētu šo informāciju vākt un saglabāt failā kādā zināmā formātā (piemēram, CSV). Windows sistēmām šim mērķim ir dabiski izmantot PerfMon rīku, kas ir operētājsistēmas sastāvdaļa. Linux sistēmām var izmantot komandu top vai mpstat automātisko periodisku izpildīšanu.

Testa darbināšana

Šajā posmā gatavs darbināšanai tests tiek palaists. Atkarībā no konfigurācijas tas var ilgt no dažām minūtēm līdz vairākām stundām vai pat dienām. Šī posma rezultātā rodas vairāki sakrāto datu faili, kurus ir iespējams analizēt.

Rezultātu analīze

Slodzes testa veikšanas laikā Picus izveido divus failus ar savāktiem veiktspējas rādītājiem. Papildus, var būt pieejami arī faili no sistēmu pārraudzības rīkiem. Šī informācija tagad ir jāliek kopā un jāanalizē tajos esošā informācija.

Tā kā šajos failos informācija ir pieejama lielā daudzumā un skaitļu formā, analizēt to ir grūti, tāpēc ir lietderīgi to pārvērst grafiskajā formā. To var darīt, piemēram, ar Excel rīku. Rezultātu analīze kā tāda, ieskaitot sistēmas šauro vietu noteikšanu, konstatēto problēmu cēloņu noteikšanu, sistēmas veiktspējas optimizācijas ceļu noteikšanu utt., ir ļoti sarežģīts process, kas varētu pati par sevi veidot atsevišķa darba tēmu. Tāpēc šinī darbā tas netiek skatīts.

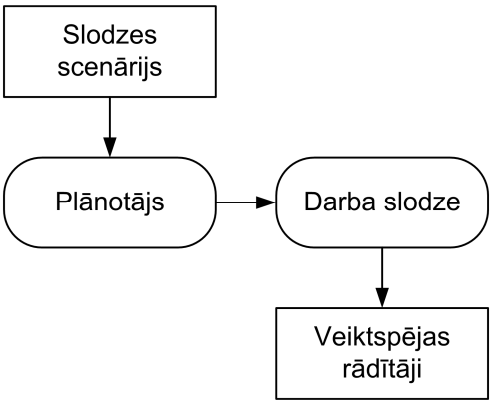
5.2. Adaptīvas veiktspējas testēšanas metodes

Lai varētu izmērīt veiktspējas rādītājus, veiktspējas tests emulē vairāku lietotāju vienlaicīgu darbu, kuri veic savas ikdienas darbības. Atkarībā no testa mērķiem var tikt imitētas dažādas noslodzes ar testa slodzes scenāriju palīdzību. Slodzes scenāriju projektēšana pirms testa veikšanas ir mūsdienās parastā prakse. Šeit tiks piedāvāta alternatīva metode, kur slodzes scenārijs tiek veidots automātiski testa laikā, lai pēc iespējas ātrāk sasniegtu testa mērķi.

5.2.1. Fiksēti slodzes scenāriji

Klasiskā pieeja darba slodzes veidošanai veiktspējas testos ir slodzes scenāriju projektēšana pirms testa darbināšanas. Gan skriptu, gan slodzes scenāriju rūpīga projektēšana ir svarīga testa mērķu sasniegšanai [110]. Šajā pieejā slodzes scenārija specifikācija kalpo par ieeju slodzes aģenta komponentam, kuru mēs sauksim par plānotāju (*scheduler*), kas ir atbildīgs par slodzes scenārijam atbilstošas darba slodzes nodrošināšanu. Veiktspējas rādītāji tiek mērīti un veido testa izvadi. Šis process ir ilustrēts 5.4. attēlā.

Vairākos gadījumu fiksēta slodzes scenārija projektēšana un izmantošana ir pietiekama un pat ir labākā metode. Piemēram, ja mērķis ir novērtēt dažādas veiktspējas metrikas pie dažādiem slodzes līmeņiem, būtu jāuzprojektē slodzes scenārijs, kurā virtuālo lietotāju skaits pakāpeniski palielinās no nulles līdz nepieciešamai testējamās sistēmas kapacitātes, jāizpilda šis tests un jāapstrādā rezultāti, lai iegūtu savāktus veiktspējas rādītājus kā funkcijas no virtuālo lietotāju skaita [6]. Šādu funkciju esamība un to reprezentēšana grafiskā formā var būt ļoti noderīga testējamās sistēmas šauro vietu noteikšanai, kā arī novērtēšanai, vai reālā testējamās sistēmas kapacitāte atbilst prasītajai.



5.4. att. Tests ar fiksētu slodzes scenāriju

Šādas pieejas trūkums ir relatīvi ilgs laiks, kas ir nepieciešams testa darbināšanai. Ja mērķis ir šaurāks, nekā piemērā minētais, tas var būt neefektīvi. Testa izpildei nepieciešamais laiks ir viens no svarīgākajiem faktoriem, kas ietekmē testa atkārtotas izpildes biežumu izstrādes laikā. Dažās programmatūras izstrādes metodēs ir nepieciešama bieža testu atkārtota izpilde un veiktspējas testi arī varētu būt iekļauti testu kopā, kas ir jāizpilda pēc katras nelielas koda izmaiņas, piemēram, ar testiem vadāmajā izstrādē [67]. Ilgi izpildes laiki varētu atturēt izstrādātājus no veiktspējas testu iekļaušanas šajā testu kopā, tāpēc tos izpildītu reti un veiktspējas problēmu atklāšana varētu noteikt ilgi pēc to ieviešanas. Tas, savukārt, nozīmētu, ka šādu par vēlu atklātu problēmu novēršana varētu kļūt ievērojami dārgāka.

Plaša veiktspējas testu mērķu klase, kurai fiksēti slodzes scenāriji ir neefektīvi, ir definējama kā nepieciešamība noteikt slodzes līmeni, pie kura testējamā sistēma nonāk kādā stāvoklī S. Daži šādu mērķu piemēri ir:

- pie kāda slodzes līmeņa vismaz vienas transakcijas vidējie izpildes laiki kļūst ilgāki par 10 sekundēm?
- pie kāda slodzes līmeņa vidēja lietojumu servera procesoru noslodze kļūst lielāka par 80%?
- pie kāda slodzes līmeņa tiek sasniegta maksimālā testējamās sistēmas kapacitāte?
- pie kāda slodzes līmeņa testējamā sistēma sāk strādāt nekorekti?

Šādi mērķi acīmredzami atšķiras no mērķa izteikt veiktspējas rādītājus kā funkciju no virtuālo lietotāju skaita:

$$p = F(v), \tag{5.1}$$

kur

p — veiktspējas rādītāja vērtība;

v — virtuālo lietotāju skaits;

F — pētāmā funkcija.

Ja mērķis ir šādas funkcijas izpēte, ir jānovērtē p vērtības visiem argumentiem noteiktā intervālā $[0, v_{\max}]$. Atšķirībā no šī mērķa piemēros dotie jautājumi pieprasa noteikt v_0 dotajam p_0, tādu, ka $F(v_0) \geq p_0$ un $F(v_0 - 1) < p_0$ (gadījumā ja F ir nedilstoša).

Lai sasniegtu šāda tipa mērķus ar fiksētiem slodzes scenārijiem ir nepieciešams izpētīt funkciju $F(v)$ slodzes līmeņiem intervālā $[v_{\min}, v_{\max}]$, kurā ir sagaidāma mērķa punkta v_0 parādīšanās. Jo šaurāks ir intervāls, jo mazāk laika ir nepieciešams testēšanai, bet arī lielāks risks izvēlēties nepareizu intervālu.

Cita pieeja, kura arī ir vairāk noderīga ja iepriekš nav droši zināms pietiekami šaurs intervāls, kuram pieder v_0, ir darbināt vairākus testus ar slodzes scenārijiem, kuriem ir dažādi slodzes palielināšanas soļi. Piemēram, pirmajā scenārijā slodze varētu palielināties uzreiz par 100 virtuālajiem lietotājiem, un būtu iespējams noteikt intervālu ar garumu $v_{\max} - v_{\min} = 100$, kurā ir sagaidāms punkts v_0. Tad varētu darbināt testu ar mazāku slodzes palielināšanas soli, lai iegūtu šaurāku intervālu un atkārtot šīs iterācijas, kamēr netiks noteikta vērtība v_0. Taču šādai pieejai ir nepieciešams daudz manuālā slodzes scenāriju pārkonfigurēšanas darba un tā joprojām nav efektīva no laika viedokļa.

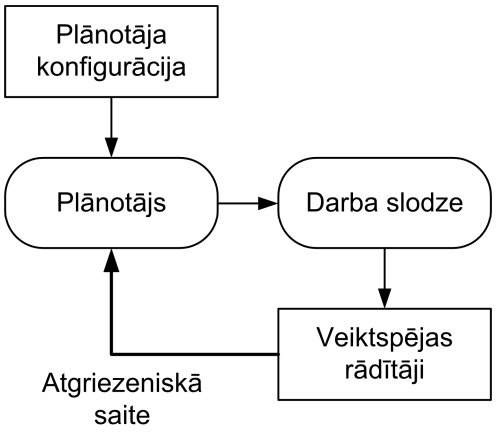
Nākamajā sadaļā tiek piedāvāta cita metode, kurā plānotājs automātiski adaptē slodzes scenāriju, lai ātrāk noteiktu v_0 punktu.

5.2.2. Adaptīvie slodzes scenāriji

Ir zināmas mainīgā darba slodzes satura dinamiskas ģenerēšanas metodes, piemēram, automātiska transakciju secības ģenerēšana no uz formām orientētiem modeļiem [31]. Šeit tiek piedāvāta metode dinamiskajai slodzes līmeņu adaptēšanai testa laikā ar mērķi panākt ātrāku testa mērķu sasniegšanu.

Adaptīvo slodzes scenāriju principi

Metode ir ilustrētā 5.5. attēlā.



5.5. att. Adaptīva slodzes scenārija modelis

Šajā modelī plānotājs nav tikai iepriekš specificētā slodzes scenārija izpildītājs, bet ir aktīvs lēmumu pieņēmējs par to, ko darīt tālāk [116]. Lēmumi, ko pieņem plānotājs ir atkarīgi no diviem faktoriem.

1. Plānotāja parametru konfigurācija. Šādi parametri nosaka mērķus, kuri ir jāsasniedz testa laikā un kurus plānotāja algoritms ir spējīgs apstrādāt.
2. Veiktspējas rādītāji, kas tika mērīti pirms lēmuma pieņemšanas brīža. Plānotājs var analizēt iepriekš iegūtos mērījumus un var balstīt uz tiem slodzes izmaiņas lēmumus.

Galvenā šī modeļa atšķirība no fiksēta slodzes scenārija modeļa ir atgriezeniska saite, ko iegūst plānotājs veiktspējas rādītāju formā.

Adaptīvā plānotāja algoritms

Zemāk tiks aprakstīts vienkāršs universāls adaptīvā plānotāja algoritms, kas var būt noderīgs dažādu veiktspējas testēšanas mērķu sasniegšanai. Bez vispārīguma zaudēšanas var pieņemt, ka funkcija $F(v)$ ir monotoni nedilstoša. Algoritms sastāv no divām fāzēm.

Pirmās fāzes mērķis ir intervāla noteikšana, kuram pieder punkts v_0. Lai tas varētu notikt ātri, lietotāju skaits pieaug eksponenciāli laikā, iteratīvi divkāršojot virtuālo lietotāju skaitu.

1. Sākt ar 1 virtuālo lietotāju ($v = 1$) un nomērīt $F(v)$.
2. Kamēr $F(v) < p_0$ divkāršot virtuālo lietotāju skaitu ($v := 2v$) un atkārtoti nomērīt $F(v)$.
3. Kad tiek sasniegta v vērtība, pie kuras $F(v) \geq p_0$, var uzskatīt, ka v_0 pieder intervālam $[v/2, v]$.

Otrās fāzes mērķis ir sašaurināt atrasto intervālu $[v_{\text{low}}, v_{\text{high}}]$:

1. Ja $v_{\text{high}} - v_{\text{low}} > 1$ punktam $v \coloneqq v_{\text{low}} + (v_{\text{high}} - v_{\text{low}})/2$ nomērīt $F(v)$.
2. Ja $F(v) \geq p_0$, tad piešķiram $v_{\text{high}} \coloneqq v$, savādāk $v_{\text{low}} \coloneqq v$.
3. Atkārtot no soļa (1) līdz $v_{\text{high}} - v_{\text{low}} = 1$. Šajā brīdī v_{high} arī ir meklējamā v_0 vērtība.

Citiem vārdiem sakot, otrajā fāzē tiek pielietots binārās meklēšanas algoritms, lai atrastu v_0 intervālā $[v_{\text{low}}, v_{\text{high}}]$. Kopējo funkcijas $F(v)$ novērtējumu skaitu n aprakstītajam algoritmam var novērtēt ar formulu:

$$n = 2\lceil \log_2 v_0 \rceil, \tag{5.2}$$

kur

v_0 — meklējamais virtuālo lietotāju skaits.

Tā kā funkcijas $F(v)$ novērtējumu skaits ir logaritmiski atkarīgs no v_0, šāda algoritma efektivitāte ir augsta, salīdzinot ar fiksēta slodzes scenārija pieeju. Piemēram, ja meklējamais slodzes līmenis ir ap 1000 virtuālo lietotāju, ir nepieciešami 20 funkcijas $F(v)$ novērtējumi jeb 20 slodzes mainīšanas soļi.

Lai algoritms varētu būt efektīvs, ir jāņem vērā daži citi aspekti:

- Pēc slodzes palielināšanas plānotājam ir jāgaida, kamēr visi jauni virtuālie lietotāji iestrādājas parastajā darba ritmā un tikai tad jāmēra nepieciešamie vidējie veiktspējas rādītāji. Gaidīšanas laikam jābūt vismaz divas reizes ilgākam, nekā aizņem viena skripta izpilde.
- Pēc slodzes samazināšanas plānotājam ir jāgaida, kamēr visi apturētie virtuālie lietotāji korekti pabeidz savas darbības un sistēma stabilizējas pēc noslodzes, ko ģenerēja šie lietotāji, un tikai tad ir jāmēra nepieciešamie vidējie veiktspējas rādītāji.

Ar skripta izpildes laiku ir saprotama laika, kas ir nepieciešams visu skripta transakciju izpildei, un visu aizturu starp transakcijām summa. Šādas piesardzības ir nepieciešamas, lai labāk nomērītu slodzes efektu un novērstu troksni mērāmajos datos, ko var radīt slodzes izmaiņu efekti.

Ir nepieciešams norādīt, ka aprakstītais algoritms varētu nenostrādāt gadījumā, ja ir citi faktori, kas var ietekmēt veiktspējas rādītājus vairāk, nekā slodzes līmenis, ko ģenerē slodzes aģents. Tādā gadījumā funkcijā $F(v)$ var būt stohastiski fluktuējoša un pieņēmums, ka tā ir monotoni nedilstoša, var būt aplams.

Laika palielināšana, kurā plānotājs savāc veiktspējas rādītājus noteiktam slodzes līmenim, varētu palīdzēt samazināt šādu faktoru ietekmi, bet tā nevar pilnīgi tos novērst. Tādā gadījumā var izrādīties, piemēram, ka nomērītie izpildes laiki pie 30 lietotāju noslodzes, varētu izrādīties lielāki, nekā pie 31 lietotāja noslodzes.

Kaut arī šādu trokšņu varbūtība ir maza pietiekami ilgiem mērīšanas laika periodiem, šādas situācijas ir iespējamas un tas fakts ir jāņem vērā.

5.2.3. Adaptīvā plānotāja algoritma eksperimentālā pārbaude

Šeit tiks aprakstīts eksperimentāls pētījums, kas ilustrē piedāvāta algoritma izmantojamību.

Eksperimentā izmantotie testi

Eksperimentālajam pētījumam kā testējamā sistēma tika izmantota uz tīmekli balstīta defektu pārvaldības sistēma Mantis.

Testā tika izmantots viens vienkāršs skripts, kas saturēja šādas transakcijas:

1) pirmās lapas atvēršana;
2) pieteikšanās;
3) defekta pieteikšanas formas atvēršana;
4) aktīvo defektu saraksta atvēršana;
5) atteikšanās.

Par testa mērķi tika izvirzīta slodzes līmeņa (vienlaicīgi strādājošo virtuālo lietotāju skaita) noteikšana, pie kura vidējais ilgākas transakcijas izpildes laiks kļūst lielāks par trim sekundēm.

Testa izstrādei un izpildei šim pētījumam tika izmantots autora izstrādātais veiktspējas testēšanas rīks Picus [114]. Picus nodrošina ērtu mehānismu alternatīvu plānotāju izveidei spraudņu formā, kas bija nepieciešama prasība pret rīku šī eksperimenta nolūkiem.

Izmantojot dažādus plānotājus tika sagatavoti divi testi:

1. Parastais pakāpeniskas palielināšanas plānotājs. Šis plānotājs strādā pēc fiksēta slodzes scenārija, šajā gadījumā sakot ar vienu lietotāju un palielinot lietotāju skaitu par vienu katrā minūtē. Maksimālais lietotāju skaits, kas bija jāsasniedz, tika specificēts kā 100, tātad testam būtu jāilgst 100 minūtes, lai izpildītos.

2. Jauns izstrādātais plānotājs, kas balstās uz aprakstīto 5.2.2. sadaļā algoritmu. Šis plānotājs novērtē funkciju $p = F(v)$, kas šajā gadījumā ir vidējais ilgākās transakcijas izpildes laiks pie lietotāju skaita v. Tātad algoritma mērķis ir noteikts kā v_0 noteikšana priekš $p_0 = 3$ sekundes. Viena slodzes palielināšanas vai samazināšanas soļa ilgums tika nokonfigurēts kā viena minūte.

Rezultāti

Divos testos iegūtie rezultāti izrādījās pietiekami līdzīgi. Tests 1 guva rezultātu $v_0 = 53$ virtuālie lietotāji, bet testa 2 rezultāts bija $v_0 = 51$ virtuālais lietotājs. Šos divus rezultātus var uzskatīt par ļoti tuviem, jo veiktspējas testēšanas stohastiskas dabas dēļ starpība starp pat diviem viena un tā paša testa laidieniem varētu būt daudz lielāka.

Starpība divos testos bija izpildes laikā. Tests 1 izpildījās 100 minūtes, kā arī bija plānots. Tests 2 izpildījās 12 minūtes, kā to varētu novērtēt, izmantojot formulu 2 un ņemot vērā to, ka slodze mainās reizi minūtē. Rezultāti ir apkopoti 5.1. tabulā.

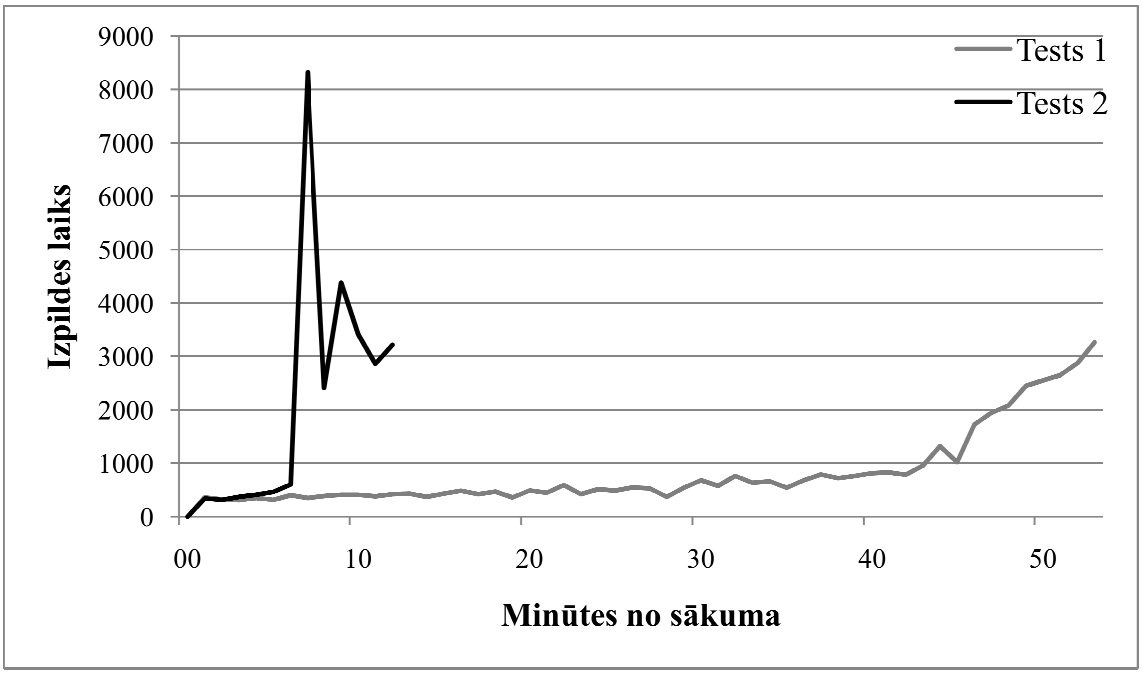
5.1. tabula

Testu rezultāti

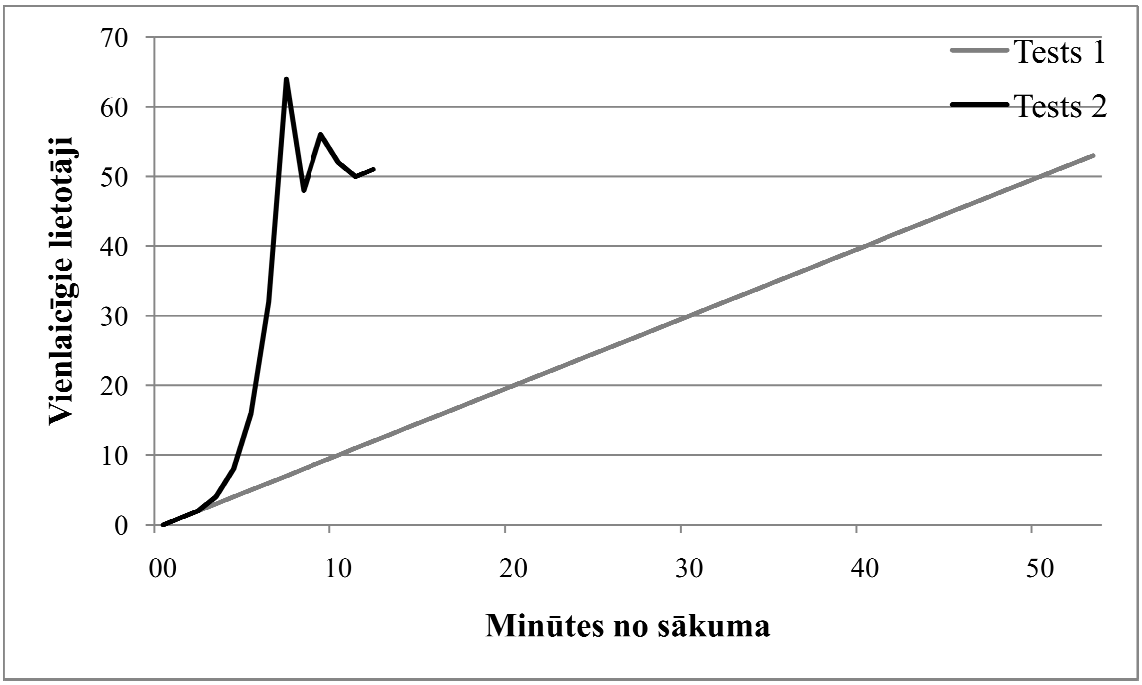
	Tests 1	Tests 2
Iegūtais slodzes līmenis	53	51
Izpildes laiks (minūtes)	100	12
Laiks līdz mērķa iegūšanai	53	12

Lai ilustrētu izpildīto testu dinamisko aspektu, tālāk ir ievietoti divi grafiki, kas parāda virtuālo lietotāju skaitu un izpildes laiku izmaiņas laika gaitā abiem testiem. Laika skala abos grafikus ir saīsināta līdz 53 minūtēm, t.i., līdz punktam, kad abi testi atklāja meklējamo v_0 vērtību.

Vidējā ilgākas transakcijas izpildes laika izmaiņas pēc izpildes laikā ir parādītas 5.6. attēlā. Slodzes līmeņa izmaiņas pēc izpildes laika ir atspoguļotas 5.7. attēlā.



5.6. att. Izpildes laiki pēc laika



5.7. att. Slodzes līmenis pēc laika

Var novērot, ka tests 2 ātri sasniedza slodzes līmeni, kad transakciju izpildes laiki ir daudz ilgāki, nekā tests 1 vispār ir ieguvis. Augstākais izpildes laiks, kas bija lielāks par 8000

milisekundēm ir moments kad tika beigta algoritma pirmā fāze. Pēc tā momenta grafika fluktuējoša daļa atbilst algoritma otrajai daļai, kad tiek veikta mērķa vērtības binārā v_0 meklēšana.

Šajā grafikā ir labi redzamas testa 2 eksponenciālas slodzes palielināšanas priekšrocības, salīdzinot ar lineāro slodzes palielināšanu testā 1.

Skatoties uz šiem grafikiem, var arī novērot, ka tests 2 dod tikai tik daudz informācijas, cik ir nepieciešams testa mērķim. No otras puses, tests 1 izdod vairāk informācijas, kas varētu noderēt dziļākai veiktspējas analīzei. Testam 1 atbilstoša līnija 5.6. attēlā pēc formas ir ekvivalenta funkcijas $F(v)$ grafikam (un, pateicoties vienas minūtes pieauguma intervālam, tā ir arī ekvivalenta pēc mēroga). No šī grafika var secināt, ka sākot ar 45 virtuālo lietotāju radītu slodzi, izpildes laiki sāk palielināties ātrāk, kas var būt liecinājums tam, kāds no sistēmas resursiem pie šī līmeņa tika izsmelts. No testa 2 grafika šādu vērtīgu (bet no testa mērķa viedokļa nerelevantu) informāciju nevarētu iegūt.

5.3. Piektās nodaļas secinājumi

5.3.1. Izpētītas veiktspējas rīku īpašības un tipiskie uzdevumi un izstrādāts veiktspējas testa modelis. Balstoties uz šo modeli ir iespējams analizēt esošos vai izstrādāt jaunus veiktspējas testēšanas rīkus.

5.3.2. Balstoties uz pētījumu, izstrādāts pielāgojams veiktspējas testēšanas rīks Picus, kas ir izmantojams dažāda veida veiktspējas testēšanas uzdevumu veikšanai un ir pielāgojams specifiskām testējamās sistēmas vai projekta īpašībām. Pēc 1.3. sadaļā izstrādātās klasifikācijas Picus pieder klasēm D3 (atbalsta datu tabulas), S1a-Java (izmanto Java programmēšanas valodu skriptu izveidei), M2 (atbalsta paralēlu skriptu izpildi), I3-HTTP (strādā protokola līmenī, pagaidām atbalstot tikai HTTP).

5.3.3. Izstrādāts adaptīvā slodzes scenārija modelis un adaptīvā plānotāja algoritms. Adaptīvs plānotājs ir implementēts kā rīka Picus sastāvdaļa. Tas ļauj ātrāk izpildīt veiktspējas testus, ja testēšanas mērķis ir specifiski uzdots.

5.3.4. Eksperimentāli parādīts, ka atsevišķiem uzdevumiem adaptīvā plānotāja pielietošana ļauj panākt būtiski labāku laika efektivitāti, samazinot testa izpildes laiku no lineārā līdz logaritmiskajam.

6. IZSTRĀDĀTO RISINĀJUMU IEVIEŠANA

Šajā nodaļā tiek īsi apkopota informācija par iepriekšējās divās nodaļās aprakstīto risinājumu lietošanu reālos projektos, sniedzot testēšanas ārpakalpojumus. Pasūtītāji šajos projektos bija vairākas Latvijas bankas un valsts iestādes, testētas tika lielas sistēmas ar tīmekļa saskarni, kuras raksturo liels lietotāju skaits.

6.1. Testu komplektu ģenerēšanas risinājumu ieviešana

Darba gaitā izstrādātie testu komplektu ģenerēšanas risinājumi un bibliotēka TSGL tika izmantota divos SIA „Jauno Tehnoloģiju Centrs" testēšanas projektos. Bibliotēka TSGL tika izmantota tikai divas reizes tāpēc, ka tajā nodrošinātas iespējas kļūst nepieciešamas tikai ļoti netriviālajos gadījumos. Vienkāršajos testu automatizācijas uzdevumos parasti pietiek ar statiskajiem testu datiņiem un TSGL lietošana nevajadzīgi sarežģītu testu programmatūru.

Taču vienā testēšanas projektā automatizācijas uzdevums izrādījās pietiekami netriviāls, lai TSGL ieviešana dotu pozitīvu efektu. Uzdevuma būtība ir šāda: bija jāpārbauda, vai lietotājiem ir tiesības veikt tās un tikai tās darbības sistēmā, kas atbilst viņa lomai (tiesību līmenim) sistēmā. Uzdevuma sarežģītību raksturo šādi lielumi:

- pārbaudāmo darbību skaits: ap 120;
- lomu skaits: ap 45.

Sistēmas integritātes kritiskuma dēļ, bija jāpārbauda visas lomu un darbību kombinācijas. Vienas darbības izpildes pārbaudei vienai lietotāja lomai manuālās testēšanas gadījumā aizņemtu 10 minūtes, tāpēc visu kombināciju pārbaude prasītu 900 cilvēkstundas jeb 5,6 cilvēkmēnešus.

Automatizējot testus, bija iespējams samazināt vienas kombinācijas testēšanai nepieciešamo laiku līdz aptuveni vienai minūtei. Tādā gadījumā pilna testa izpilde aizņemtu 90 stundas, kas joprojām nebija pieņemami, jo tika sagaidīts, ka tiks atklātas neatbilstības, tās tiks labotas un pilnais tests būs jāizpilda atkārtoti. Četras diennaktis ilgs tests nebija pieņemams.

Vienkāršojot testu skriptus un uzticot TSGL testu komplektu ģenerēšanu, pilnajam testam nepieciešamais laiks samazinājās līdz 30 stundām. Paralelizējot testa izpildi uz trīs datoriem, ir izdevies testa kopējo laiku samazināt līdz 10 stundām, kas jau bija pieņemami.

Atklāto neatbilstību dēļ tests bija jāatkārto 4 reizes (notika četri testēšanas-labošanas cikli), tāpēc uz TSGL balstītās pieejas izmantošana attaisnojās.

Šajā testā tika izmantots uz saliktajiem stāvokļiem balstītais komplektētājs, un stāvokļa komponenti bija šādi:

1) lietotāja loma — tiesību līmenis, kas ir tekošajam sistēmas lietotājam;
2) aktīvais logs — kurš logs sistēma dotajā brīdī aktīvs.

Būtiskākā TSGL loma bija tāda, ka testu skripti varēja neiekļaut darbības, kas atver nepieciešamo logu, un beigās atgriežas sākuma stāvoklī — pietika norādīt, kurā logā katram testam ir jāsākas, un kurā jābeidzas — pareizo izpildes secību TSGL izvēlējās automātiski, līdz ar to tika panākta laika ekonomija uz liekas logu atvēršanas un aizvēršanas. Bez TSGL tas arī būtu iespējams, taču tas sarežģītu skriptu izstrādi ar to, ka secība būtu jāveido manuāli, un manuāli jāpārveido pēc skriptu atjaunošanas.

Otrajā projektā, kur tika izmantota bibliotēka TSGL, tā tika lietota saiknē ar Picus, nodrošinot virtuālā lietotāja darbību secības ģenerēšanu. TSGL izmantošanas lietderība bija pamatota ar to, ka veicamo darbību skaits projektā bija liels (ap 50) un darbību veikšanas iespējamība lielā mērā bija atkarīga no tekoša sistēmas stāvokļa. Taču šajā gadījumā TSGL izmantošanas atdeve nebija tik liela kā pirmajā projektā, jo darbību skaits tomēr bija ievērojami mazāks.

6.2. Veiktspējas testēšanas rīka Picus ieviešana

Kopš pirmās versijas izveides un līdz šī darba rakstīšanas brīdim Picus tika izmantots 13 SIA „Jauno Tehnoloģiju Centrs“ testēšanas projektos. Visi projekti bija saistīti ar tīmekļa sistēmu veiktspējas testēšanu, līdz ar to tika izmantots esošais Picus modulis HTTP protokola atbalstam. Lai ilustrētu projektu dažādību, četri no tiem tiek aprakstīti nedaudz detalizētāk:

1) Testējamā sistēma bija izstrādāta, izmantojot valodu Java un ietvaru Struts. Tomcat bija izmantojams kā lietojumu konteineris un tīmekļa serveris. Tika izstrādāti divi Picus skripti. Vienam skriptam bija nepieciešams sūtīt sistēmai unikālus specifiska formāta datus, tāpēc lai nodrošinātu to iespēju, Picus tika paplašināts ar datu tabulu atbalstu. Realizācijā datu tabulām ir jābūt CSV formas failiem, no kuriem skripts ņem vienu kārtējo rindu, kad tas ir nepieciešams. Pietiekami lielas datu tabulas garantē, ka skripti katru reizi izmanto savu datu komplektu.

2) Testējamā sistēma bija izstrādāta, izmantojot Oracle rīkus, un Oracle Web Application Server tika izmantots kā tīmekļa serveris. Tika izstrādāti pieci skripti, kas izpildīja piecus dažādus datu atlasīšanas scenārijus. Pieprasījumos tika izmantoti lieli datu apjomi ar neparastu kodējumu, tāpēc uz šī projekta precedenta Picus HTTP protokola modulī tika ieviests šādu pieprasījumu atbalsts.
3) Testējamā sistēma bija izstrādāta, izmantojot Ruby-on-Rails un Microsoft Infopath tehnoloģiju kombināciju. MS IIS tika izmantots kā tīmekļa serveris. Lai nodrošinātu darba ar Infopath emulāciju, bija nepieciešams rīkam Picus pievienot iespēju apmainīties ar XML datiem caur HTTP protokolu. Līdz tam brīdim esošais sīkdatņu (*cookies*) apstrādes mehānisms nebija savietojams ar testējamo sistēmu, tāpēc tika uzlabotas sīkdatņu apstrādes iespējas HTTP protokola modulī.
4) Sistēma bija izstrādāta, izmantojot valodas Java un JSP. Tomcat bija izmantots kā lietojumu konteineris un tīmekļa serveris. Šai sistēmai jau nebija nepieciešams paplašināt Picus iespējas — pietika ar jau esošo funkcionalitāti.

Kā var redzēt no šiem piemēriem, reālu projektu veikšanas rezultātā vairākos aspektos tika uzlabots HTTP protokola modulis. Pašlaik tas ir pietiekami stabils, lai bez izmaiņām varētu atbalstīt gandrīz jebkuru tīmekļa sistēmu testēšanu.

Visi minētie, kā arī citi projekti bija sekmīgi izpildīti un aizņēma no vienas līdz trim nedēļām, skaitot no testēšanas prasību fiksēšanas brīža līdz rezultātu kopsavilkuma atskaites sagatavošanai. Tehniskais testu sagatavošanas darbs, ieskaitot skriptu izstrādi, Picus pielāgošanu un testa parametru konfigurēšanu, aizņēma no divām līdz piecām darba dienām. Gūta pieredze rāda, ka rīks Picus ir pietiekami labi atbilst savam nolūkam. Laika daudzums, kas bija jāpatērē šiem projektiem ir tuvs laikam, kas bija jāvelta līdzīgiem projektiem, izmantojot citus rīkus, vai pat labāks. Tas ir izskaidrojams ar Picus pielāgojamību pat pietiekami netriviālām situācijām.

6.3. Sestās nodaļas secinājumi

6.3.1. Izstrādātā TSGL bibliotēka tika sekmīgi lietota divos testēšanas projektos, kuros nodrošināja automātisku testu komplektu ģenerēšanu no lielām testu kopām. TSGL izmantošana ļāva samazināt testēšanas laiku 12 reizes, salīdzinot ar tradicionālo automatizēto testēšanu.

6.3.2. Izstrādātais veiktspējas testēšanas rīks Picus tika sekmīgi izmantots 13 testēšanas projektos austi noslogotu tīmekļa sistēmu veiktspējas testēšanai. Veiktspējas testa sagatavošanas laiks rīkam Picus bija no divām līdz piecām darba dienām, bet izpildes vienai testa izpildei svārstījās no 15 minūtēm līdz 3 stundām.

NOBEIGUMS

Darba mērķis bija izstrādāt programmatūras testēšanas automatizācijas metodes, kas ļautu padarīt automatizēto testēšanu efektīvāku. Darba rezultātā tika izstrādātas metodes automatizētai testu komplektu ģenerēšanai, kā arī adaptīvajai veiktspējas testēšanai, kas tika aprobēti reālos projektos.

Būtiskākie darba sasniegumi.

1. Izstrādāta automatizētās testēšanas rīku klasifikācija, pēc kuras tika klasificēti 32 testu automatizācijas rīki, kas ļauj izvēlēties piemērotāko rīku konkrētas programmatūras testēšanai.
2. Tika izstrādāts vienots automatizētās testēšanas modelis, kas ļauj identificēt potenciāli automatizējamos procesus programmatūras testēšanā. Modelis ir piemērojams dažādos testu automatizācijas kontekstos un dažādi implementējams atkarībā no rīka klases.
3. Tika izstrādāts manuālo un automatizēto testu efektivitātes novērtēšanas matemātiskais modelis. Uz modeļa pamata ir izstrādāts efektīvās testu kopas izvēles algoritms, kas ir izmantojams, lai balstoties uz defektu risku un testēšanas darbietilpības novērtējumiem izvēlēties testus, kas ar lielāku varbūtību atklās vairāk defektu ierobežotā laikā.
4. Tika izstrādātas divas automatizētās testu komplektu ģenerēšanas metodes, balstītās uz testējamu sistēmu stāvokļiem: vienkāršajiem un saliktajiem. Metodes tika implementētas bibliotēkā TSGL, kas tika aprobēta divos reālos projektos.
5. Tika izstrādāts veiktspējas rīks Picus, īpašs ar tā elastību un paplašināšanas iespējām. Picus rīks tika aprobēts 13 reālos projektos.
6. Tika izstrādāta adaptīvās veiktspējas testēšanas metode, kas tika realizēta kā Picus rīka spraudnis.

Izstrādātajiem risinājumiem ir tāda praktiskā nozīme, ka tos ir iespējams izmantot programmatūras izstrādes projektos automatizētās testēšanas efektivitātes uzlabošanai. Izstrādātie rīki un pieredze to izmantošanā projektos liecina, ka izstrādātie risinājumi ir reāli ieviešami un praktiski noderīgi.

LITERATŪRA

1. Abbas N., Gravell A. M., Wills G. B. Historical Roots of Agile Methods: Where did "Agile Thinking" come from? // Agile Processes in Software Engineering and Extreme Programming. – 2008. – pp. 94–103.
2. Abbot. Abbot framework for automated testing of Java GUI components and programs. / Internets. – http://abbot.sourceforge.net/ (skatīts 29.05.2011)
3. Adrion W., Branstad M., Cherniavsky J. Validation, Verification, and Testing of Computer Software // ACM Computing Surveys. – 1982. – Vol. 14, No. 2. – pp. 159–192.
4. Aichernig B. et al. State of the Art Survey - Part a: Model-based Test Case Generation. Technical Report 1-19a on project MOGENTES. – Graz University of Technology. – 2008.
5. Amland S., Garborgsv H. Risk Based Testing and Metrics // 5th International Conference EuroSTAR. – 1999. – pp. 1–20.
6. Amza C. et al. Specification and implementation of dynamic web site benchmarks // 5th IEEE Workshop on Workload Characterization (WWC-5). – IEEE Press, 2002. – pp. 3–13.
7. Angļu-latviešu-krievu informātikas vārdnīca. – Rīga: Avots, 2001. – 660 lpp.
8. Apache Jakarta Project. Apache JMeter. / Internets. – http://jakarta.apache.org/jmeter/ (skatīts 29.05.2011)
9. Arnicans G., Arnicane V. Using the Sponsor-User-Programmer Model to Improve the Testing Process // Scientific Papers University of Latvia. – 2009. – Vol. 751. – pp. 65–79.
10. Ash L. The Web Testing Companion: The Insider's Guide to Efficient and Effective Tests. – USA: John Wiley & Sons, 2003. – 600 p.
11. AutomatedQA. TestComplete. / Internets. – http://www.automatedqa.com/products/testcomplete/ (skatīts 29.05.2011)
12. Baresi L., Young M. Test Oracles. Technical Report CIS-TR-01-02. – University of Oregon, Dept. of Computer and Information Science, Eugene, OR, USA. – 2001. – 55 p.

13. Basili V. R., Perricone B. T. Software Errors and Complexity: An Empirical Investigation // Communications of the ACM. – 1984. – Vol. 27, No. 1. – pp. 42–52.
14. Beck K. Test Driven Development: By Example. – Boston, MA, USA: Addison-Wesley, 2002. – 240 p.
15. Beck K., Andres C. Extreme Programming Explained: Embrace Change. 2nd ed. – Boston, MA, USA: Addison-Wesley Professional, 2004. – 224 p.
16. Beizer B. Software Testing Techniques. 2nd ed. – NY, USA: Van Nostrand Reinhold, 1990. – 550 p.
17. Beust C., Suleiman H. Next Generation Java Testing: TestNG and Advanced Concepts. – Boston, MA, USA: Addison-Wesley, 2007. – 512 p.
18. Bičevskis J. et al. SMOTL – A System to Construct Samples for Data Processing Program Debugging // IEEE Transactions on Software Engineering. – 1979. – Vol. SE-5, No. 1. – pp. 60–66.
19. Boehm B. W. A spiral model of software development and enhancement // IEEE Computer. – 1988. – Vol. 21, No. 5. – pp. 61–72.
20. Boehm B. W. Seven Basic Principles of Software Engineering // Journal of Systems and Software. – 1983. – Vol. 3, No. 1. – pp. 3–24.
21. Borland. SilkPerformer. / Internets. – http://www.borland.com/us/products/silk/silkperformer/ (skatīts 29.05.2011)
22. Borland. SilkTest. / Internets. – http://www.borland.com/us/products/silk/silktest/ (skatīts 29.05.2011)
23. Braberman V., Felder M., Marré M. Testing Timing Behavior of Real-Time Software // Proceedings of 10th International Software Quality Week. – San Francisco, CA, USA: Software Research Institute, 1997. – pp. 145–155.
24. Buckley F. J., Poston R. Software Quality Assurance // IEEE Transactions on Software Engineering. – 1984. – Vol. SE-10, No. 1. – pp. 36–41.
25. Canoo. Canoo WebTest. / Internets. – http://webtest.canoo.com/ (skatīts 29.05.2011)
26. Clark T., Warmer J. Object Modeling with the OCL: The Rationale Behind the Object Constraint Language. – Springer, 2002. – 281 p.
27. Cormen T. H. et al. Introduction to Algorithms. 2nd ed. – Cambridge, MA, USA: MIT Press, 2001. – 1184 p.
28. CppUnit. CppUnit Wiki. / Internets. – http://cppunit.sourceforge.net/ (skatīts 29.05.2011)

29. CUnit. A Unit Testing Framework for C. / Internets. – http://cunit.sourceforge.net/ (skatīts 29.05.2011)
30. Denaro G., Polini A., Emmerich W. Early performance testing of distributed software applications // 4th International Workshop on Software and Performance (WOSP ’04). – ACM, 2004. – pp. 94–103.
31. Draheim D. et al. Realistic load testing of web applications // 10th European Conference on Software Maintenance and Reengineering (CSMR 2006). – IEEE Computer Society, 2006. – pp. 57–70.
32. Dustin E., Rashka J., Paul J. Automated Software Testing: Introduction, Management and Performance. – Boston, MA, USA, 1999. – 608 p.
33. E-naxos. DataGen - Test Data Generator. / Internets. – http://www.e-naxos.com/datagen.aspx (skatīts 29.05.2011)
34. Edvardsson J. A survey on automatic test data generation // Proceedings of the 2nd Conference on Computer Science and Engineering. – 1999. – pp. 21–28.
35. Elvior. Elvior TestCast. / Internets. – http://www.elvior.com/testcast/introduction (skatīts 29.05.2011)
36. Fagan M. E. Design and code inspections to reduce errors in program development // IBM Systems Journal. – 1976. – Vol. 15, No. 3. – pp. 182–211.
37. Fejes B. Test Web applications with HttpUnit. / Internets. – http://www.javaworld.com/javaworld/jw-04-2004/jw-0419-httpunit.html (skatīts 29.05.2011)
38. Ferguson R., Korel B. The chaining approach for software test data generation // ACM Transactions on Software Engineering and Methodology. – 1996. – Vol. 5, No. 1. – pp. 63–86.
39. Fewster M., Graham D. Software Test Automation: Effective Use of Test Execution Tools. – New York, NY, USA: ACM Press/Addison-Wesley, 1999. – 596 p.
40. ForSQL. Data Test Generator. / Internets. – http://www.forsql.com/ (skatīts 29.05.2011)
41. Fowler M., Scott K. UML Distilled: Applying the Standard Object Modelling Language. – USA: Addisson Wesley, 1997. – 208 p.
42. Futatsugi K. et al. Principles of OBJ2 // Proceedings of the 12th ACM Symposium on Principles of Programming Languages. – 1995. – pp. 21–28.
43. Gallagher B. Three tools for testing C/S applications vary in value // PC Week. – 1994. – Vol. 11, No. 47. – pp. 89.

44. Gelperin D., Hetzel B. The Growth of Software Testing // Communications of the ACM. – 1988. – Vol. 31, No. 6. – pp. 687–695.
45. Gills M. Programmatūras testēšana un trasējamība. Promocijas darbs. – Rīga: Latvijas Universitāte, 2005. – 177 lpp.
46. Godefroid P. Compositional dynamic test generation // Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. – 2007. – pp. 47–54.
47. Graham D. et al. Foundations of Software Testing: ISTQB Certification. – London, UK: Thomson Learning, 2006. – 258 p.
48. Gupta N., Mathur A., Soffa M. Automated test data generation using an iterative relaxation method // Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering. – 1998. – pp. 231–244.
49. Hamill P. Unit Testing Frameworks. – Sebastopol, CA, USA: O'Reilly, 2004. – 304 p.
50. Holzmann G. J. The Spin Model Checker: Primer and Reference Manual. – USA: Addison-Wesley Professional, 2003. – 608 p.
51. Hong H. S. et al. A Temporal Logic Based Theory of Test Coverage and Generation // Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. – 2002. – pp. 327–341.
52. HP – BTO Software. HP Business Process Testing software. / Internets. – https: //h10078.www1.hp.com/cda/hpms/display/main/hpms_content.jsp?zn=bto&cp=1-11-127-24^135_4000_100__ (skatīts 29.05.2011)
53. HP – BTO Software. HP LoadRunner software. / Internets. – https: //h10078.www1.hp.com/cda/hpms/display/main/hpms_content.jsp?zn=bto&cp=1-11-126-17^8_4000_100__ (skatīts 29.05.2011)
54. HP – BTO Software. HP Unified Functional Testing software. / Internets. – https://h10078.www1.hp.com/cda/hpms/display/main/hpms_content.jsp?zn=bto&cp=1-11-127-24^1352_4000_100__ (skatīts 29.05.2011)
55. HttpUnit. HttpUnit Home. / Internets. – http://httpunit.sourceforge.net/ (skatīts 29.05.2011)
56. Hunt A., Thomas D. Pragmatic Unit Testing in Java with JUnit. – USA: The Pragmatic Programmers, 2003. – 176 p.
57. Hutcheson M. L. Software Testing Fundamentals: Methods and Metrics. – New York, USA: Wiley, 2003. – 432 p.

58. IBM. IBM Rational Functional Tester. / Internets.
– http://www-01.ibm.com/software/awdtools/tester/functional/ (skatīts 29.05.2011)
59. IBM. IBM Rational Performance Tester. / Internets.
– http://www-01.ibm.com/software/awdtools/tester/performance/ (skatīts 29.05.2011)
60. IBM. Rational Robot. / Internets.
– http://www-01.ibm.com/software/awdtools/tester/robot/ (skatīts 29.05.2011)
61. IEEE Std 1028-2008. IEEE Standard for Software Reviews and Audits. – 2008. – 52 p.
62. IEEE Std 610.12-1990. IEEE Standard Glossary of Software Engineering Terminology. – 1990. – 96 p.
63. ISO 8807:1989. Information processing systems – Open Systems Interconnection – LOTOS – a formal description technique based on the temporal ordering of observational behaviour. – 1989.
64. ITU-T Recommendation Z.100 (11/99). Specification and Description Language (SDL). – 2000. – 246 p.
65. Java.net. Jemmy. / Internets. – http://java.net/projects/jemmy/ (skatīts 29.05.2011)
66. Jessop W. H. et al. ATLAS – An Automated Software Testing System // Proceedings of the 2nd international conference on software engineering. – CA, USA, 1976. – pp. 629–635.
67. Johnson M. J. et al. Incorporating Performance Testing in Test-Driven Development // IEEE software. – 2007. – Vol. 24, No. 3. – pp. 67–73.
68. JUnit.org. Resources for Test Driven Development. / Internets. – http://junit.org/ (skatīts 29.05.2011)
69. Kan S. H. Metrics and Models in Software Quality Engineering. 2nd ed. – Boston, MA, USA: Addison-Wesley, 2002. – 560 p.
70. Kaner C. Architectures of Test Automation. / Internets.
– http://www.kaner.com/testarch.html (skatīts 29.05.2011)
71. Kaner C., Bach J., Pettichord B. Lessons Learned in Software Testing. – New York, NY, USA: Wiley, 2001. – 352 p.
72. Kaner C., Falk J., Nguyen H. Q. Testing Computer Software. 2nd ed. – New-York, USA: Wiley, 1999. – 480 p.
73. King J. Symbolic execution and program testing // Communications of the ACM. – 1976. – Vol. 19, No. 7. – pp. 385–394.
74. Korel B. Automated software test data generation // IEEE Transactions on Software Engineering. – 1990. – Vol. 16, No. 8. – pp. 870–879.

75. Kosinski S. The ENCORE Stress Test Generator for On-line Transaction Processing Applications // Tandem Journal. – 1984. – Vol. 2, No. 1. – pp. 6–17.
76. Kruchten P. The Rational Unified Process: An Introduction. 3rd ed. – Boston, MA, USA: Addison-Wesley Professional, 2003. – 336 p.
77. Larman C., Basili V. R. Iterative and incremental development: A brief history // IEEE Computer. – 2003. – Vol. 36, No. 6. – pp. 47–56.
78. Leavens G. T., Baker A. L., Ruby C. Preliminary design of JML: A behavioral interface specification language for Java // ACM SIGSOFT Software Engineering Notes. – 2006. – Vol. 31, No. 3. – pp. 1–38.
79. Lewis R., Beck D. W., Hartmann J. Assay – A Tool to Support Regression Testing // 2nd European Software Engineering Conference, Proceedings. – Springer, 1989. – pp. 487–496.
80. Lewis W. E. Software Testing and Continuous Quality Improvement. 3rd ed. – Auerbach Publications, 2008. – 688 p.
81. Li K., Wu M. Effective GUI Test Automation: Developing an Automated GUI Testing Tool. – Alameda, CA, USA: Sybex: Sybex, 2004. – 445 p.
82. Marick B. When should a test be automated // Proceedings of The 11th International Software/Internet Quality Week. – 1998. – pp. 1–20.
83. Massol V., Husted T. JUnit in action. – Greenwich, CT, USA: Manning Publications, 2003. – 384 p.
84. Memon A., Pollack M., Soffa M. Hierarchical GUI test case generation using automated planning // IEEE Transactions on Software Engineering. – 2001. – Vol. 27, No. 2. – pp. 144–155.
85. Meyer B. Eiffel: The Language. – Prentice Hall, 1991. – 300 p.
86. Micro Focus. Micro Focus TestPartner™: Functional testing. / Internets. – http://www.microfocus.com/products/silk/testpartner.aspx (skatīts 29.05.2011)
87. Micro Focus. QALoad. / Internets. – http://www.microfocus.com/products/silk/QALoad.aspx (skatīts 29.05.2011)
88. Mosley D. J., Posey B. A. Just Enough Software Test Automation. – Prentice Hall, 2002. – 288 p.
89. Mosses P. D. CASL: A guided tour of its design // Proceedings of WADT'98. – Springer-Verlag, 1999. – pp. 216–240.
90. Musa J. D. Software Reliability Engineering: More Reliable Software Faster and Cheaper. 2nd ed. – Bloomington, IN, USA: AuthorHouse, 2004. – 632 p.

91. Myers G. J. The Art of Software Testing. – NY, USA: John Wiley & Sons, 1976. – 177 p.
92. Näätänen J., Suhorukovs A. Compuware QACenter testēšanas pārvaldības un automatizācijas risinājumi // Latvijas IT uzņēmumu 7. konference „Testēšanas teorija un prakse“: konferences materiāli. – Rīga, Latvija, 2006. – lpp. 33. –35.
93. NUnit. NUnit - Home. / Internets. – http://www.nunit.org/ (skatīts 29.05.2011)
94. Olshefski D. P., Nieh J., Agrawal D. Inferring client response time at the web server // International Conference on Measurements and Modeling of Computer Systems (SIGMETRICS 2002). – USA: ACM, 2002. – pp. 160–171.
95. OpenSTA. OpenSTA Users Home Page - Free Web Load and Stress Testing Tool. / Internets. – http://www.opensta.org/ (skatīts 29.05.2011)
96. OpenTTCN.com. / Internets. – http://www.openttcn.com/ (skatīts 29.05.2011)
97. Oracle. Oracle Functional Testing. / Internets. – http://www.oracle.com/technetwork/oem/grid-control/overview/ds-oracle-2.pdf (skatīts 29.05.2011)
98. Oracle. Oracle Load Testing. / Internets. – http://www.oracle.com/technetwork/oem/grid-control/overview/ds-oracle-load-testing-1-129318.pdf (skatīts 29.05.2011)
99. PAMIE. Python Automation Module for I.E. / Internets. – http://pamie.sourceforge.net/ (skatīts 29.05.2011)
100. Patton R. Software Testing. 3rd ed. – Indianapolis, IN, USA: Sams, 2005. – 405 p.
101. Pfleeger S. L. Software Engineering: Theory and Practice. 2nd ed. – Prentice Hall, 2001. – 659 p.
102. Pressman R. S. Software Engineering: A Practicioner’s Approach. 4th ed. – McGraw-Hill, 1997. – 852 p.
103. QFS. QF-Test - Fatcts and Features. / Internets. – http://www.qfs.de/en/qftest/index.html (skatīts 29.05.2011)
104. Rapps S., Weyuker E. J. Selecting Software Test Data Using Data Flow Information // IEEE Transactions on Software Engineering. – 1985. – Vol. SE-11, No. 4. – pp. 367–375.
105. Rising L., Janoff N. S. The Scrum Software Development Process for Small Teams // IEEE Software. – 2000. – Vol. 17, No. 4. – pp. 26–32.
106. Royce W. Managing the Development of Large Software Systems // Proceedings of IEEE WESCON. – Los Angeles, CA, USA: IEEE Press, 1970. – pp. 1–9.

107. Runeson P. A survey of unit testing practices // IEEE Software. – 2006. – Vol. 23, No. 4. – pp. 22–29.
108. SeleniumHQ. Selenium web application testing system. / Internets. – http://seleniumhq.org/ (skatīts 29.05.2011)
109. Stottlemyer D. Automated Web Testing Toolkit: Expert Methods for Testing and Managing Web Applications. – NY, USA: John Wiley & Sons, 2001. – 304 p.
110. Subraya B. M. Integrated approach to web performance testing: A practitioner's guide. – PA, USA: IRM Press, 2006. – 368 p.
111. Suhorukovs A. Veiktspējas testēšanas rīka Picus izstrāde un pielietošanas iespējas // Latvijas IT uzņēmumu 9. konference „Testēšanas teorija un prakse": konferences materiāli. – Rīga, Latvija, 2008. – lpp. 31. –39.
112. Sukhorukov A. Architecture for Automated Validation of E-Learning Courses // Proceedings of the Ninth IEEE International Conference on Advanced Learning Technologies (ICALT-2009). – Washington, DC, USA: IEEE Computer Society, 2009. – pp. 152–153.
113. Sukhorukov A. Automated Test Chaining Based on Compound States // Proceedings of the International Conference on Information Technologies (InfoTech-2010). – Sofia, Bulgaria: Publishing House of Technical University, 2010. – pp. 127–132.
114. Sukhorukov A. Performance testing tool Picus // Proceedings of the 22nd International Conference on Systems for Automation of Engineering and Research (SAER-2008). – Bulgaria: King, 2008. – pp. 165–172.
115. Sukhorukov A. Problems of Test-Driven Aspect-Oriented Development // Scientific Proceedings of Riga Technical University. – 2009. – Vol. 38. – pp. 180–186.
116. Sukhorukov A. Self-Directed Performancd Testing // Scientific Journal of RTU. Series 5. – 2010. – Vol. 43. – pp. 84–89.
117. Sukhorukov A. Test Case Generation for Validation of E-Learning Course // Advances in Databases and Information Systems, 13th East-European Conference, ADBIS 2009 Associagted Workshops and Doctoral Consortium. Local Proceedings. – Riga, Latvia: Riga Technical University, 2009. – pp. 230–237.
118. Sukhorukov A., Zaitseva L. Applicability of Automated Testing to E-Learning Software Validation // Proceedings of the International Conference on Engineering Education & Research (ICEER 2007). – Melbourne, Australia: Victoria University, 2007. – pp. 1–6.
119. Sukhorukov A., Zaitseva L. Automated test state management framework // Scientific Proceedings of Riga Technical University. Series 5. – 2008. – Vol. 34. – pp. 215–224.

120. TestNG. TestNG. / Internets. – http://testng.org/doc/index.html (skatīts 29.05.2011)
121. The Grinder. The Grinder, a Java Load Testing Framework. / Internets. – http://grinder.sourceforge.net/ (skatīts 29.05.2011)
122. Tricentis Technology & Consulting. TOSCA Testsuite™. / Internets. – http://www.tricentis.com/tosca+M52087573ab0.html (skatīts 29.05.2011)
123. TTCN-3. Commercial TTCN-3 Tools. / Internets. – http://www.ttcn-3.org/CommercialTools.htm (skatīts 29.05.2011)
124. Van Veenendaal E. (ed.) ISTQB Standard Glossary of Terms Used in Software Testing. Version 2.1. – ISTQB, 2010. – 51 p. / Internets. – http://istqb.org/download/attachments/2326555/ISTQB+Glossary+of+Testing+Terms+2+1.pdf (skatīts 29.05.2011)
125. WAPT. Web Application Load, Stress and Performance Testing. / Internets. – http://www.loadtestingtool.com/ (skatīts 29.05.2011)
126. Willcock C. et al. An Introduction to TTCN-3. – Chichester, West Sussex, UK: Wiley, 2005. – 282 p.
127. Xia J. et al. An Empirical Performance Study on PSIM // The Computer Journal. – 2006. – Vol. 44, No. 5. – pp. 509–526.
128. Борзов Ю. В., Утранс Г. Б., Шимаров В. А. Выбор путей программы для построения тестов // Управляющие системы и машины. – 1989. – № 6. – стр. 29–36.
129. Винниченко И. Автоматизация процессов тестирования. – Санкт-Петербург: Питер, 2005. – 203 стр.
130. Сухоруков А. Целенаправленное обучение на примере модели и классификатора инструментов автоматизации тестирования ПО // Образовательные технологии и общество. – Казань, Татарстан, РФ: Казанский государственный технологический университет, 2010. – Том 13, № 1. – стр. 370–377.